

Applicant: Thomas R. Firman  
Filed: May 9, 2001  
For: VOICE CONTROLLED COMPUTER INTERFACE  
Attorney of Record: David L. Feigenbaum, Reg. No. 30,378  
Fish & Richardson P.C.  
225 Franklin Street  
Boston, MA 02110

## APPENDIX C

### CERTIFICATE OF MAILING BY EXPRESS MAIL

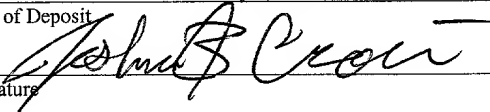
Express Mail Label No. EL298426507US

I hereby certify under 37 CFR §1.10 that this correspondence is being deposited with the United States Postal Service as Express Mail Post Office to Addressee with sufficient postage on the date indicated below and is addressed to the Commissioner for Patents, Washington, D.C. 20231.

May 9, 2001

Date of Deposit

Signature



Joshua Cronin

Typed or Printed Name of Person Signing Certificate

T06050" 64025860

[illegible]

2

```

*/
enum
{
    CW_HASFOCUS    = 1,
    CW_PARENTLEVEL = 2,
};

/*-----
|
| Description for item in context list.
|
*/
typedef struct tagCONTEXTITEM
{
    enum
    {
        // What type of context info is it.
        CON_WIND,           // It is a window or a control.
        CON_ICON,           // An iconized window.
        CON_SYSCOM,         // It is a universal window control. min/max/sys
        CON_SCROLL,         // Scrolling commands.
        CON_MENUPOPUP,      // A menu bar item that will popup.
        CON_MENU,           // A menu item in the active menu.
        CON_ACCEL,          // A short cut key.
        CON_LAUNCH,         // Executable item.
        CON_MACRO,
    } conType;

    int iLevel;             // The window group/probability level.
    HWND hwnd;             // Handle to the window associated.

    union
    {
        struct
        {
            enum
            {
                // Is it a class we know about.
                CWC_STATIC,
                CWC_BUTTON,
                CWC_LISTBOX,
                CWC_COMBOBOX,
                CWC_EDIT,
                CWC_SCROLLBAR,
                CWC_PMGROUP,
                CWC_MDICLIENT,
                CWC_CHILD,      // Other child
                CWC_GROUPBOX,   // Special case
                CWC_POPUP,      // Other popup
            } cwc;
            BOOL bForList;
            LPSTR szName;
        } Window;

        int SysCom;          // System command id.

        int ScrCom;          // Scroll Interfase
    }
};

```





```

/*-----
|
| Context List.
|
*/
_LOCAL CONTEXTITEM * pciFirst = NULL;
_LOCAL CONTEXTITEM * pciLast = NULL;

_LOCAL unsigned iChecksum = (UINT)-1;    // Keep a check sum of the context.

_LOCAL HWND hwndFocus = NULL;            // Focus window
_LOCAL HWND hwndActive = NULL;           // Active window
_LOCAL HWND hwndParent;                  // Current parent interrogated.
_LOCAL HWND hwndPrvParent;               // This was the previous parent.

_LOCAL int iCaptionLen;                  // The longest context caption length.
_LOCAL int iDebugCapLen;
_LOCAL int iGroupLevel;                  // The context group number.

_LOCAL FARPROC lpprocContext = NULL;

_LOCAL char szCaptionBuf[2 * MAX_SYMBOL_LENGTH + 50]; // Caption buffer.

_LOCAL LPLANG pLangCur = NULL;

/*-----
|
| These are switches
|
*/
_LOCAL BOOL bChildSysMenu;    // Child sys commands used ?
_LOCAL HWND hwndMenuSysPop;   // Is the sys menu popped up ?
_LOCAL BOOL bMenuBarExist;
_LOCAL BOOL bMenuPopExist;    // Is there a popup menu active.
_LOCAL int iScrollMask;       // Is scroll present mask.

/*-----
|
| These are predefined classes.
|
*/
_LOCAL PSTR szPredefClass[] =
{
    "Static",
    "Button",
    "ListBox",
    "ComboBox",
    "Edit",
    "ScrollBar",
    "PMGroup",           // Program manager groups.
    "MDIClient",
};

/*-----
|
| FUNCTION _LOCAL void ContextListInit(void)

```

DESCRIPTION Clear the previous context list.

PARAMETERS None.

RETURN None.

```
*/
_LOCAL void ContextListInit(void)
{
    /* Delete old context list
    */
    while (pciFirst != NULL)
    {
        pciLast = pciFirst->pciNext;
        if (pciFirst->conType == CON_LAUNCH)
        {
            /* We allocate these string
            */
            StringNearDestroy(pciFirst->u.PMItem.szTitle);
            StringNearDestroy(pciFirst->u.PMItem.szFile);
        }
        Nfree(pciFirst);
        pciFirst = pciLast;
    }

    /* Reset the checking environment.
    */
    iCheckSum = 0;

    /* Leave 0 for the lang overrides.
    */
    iGroupLevel = 1;

    /* A pop up menu is on top.
    */
    hwndMenuSysPop = NULL;

    /* The menu bar has been read ?
    */
    bMenuBarExist = FALSE;

    /* Child sys commands used ?
    */
    bChildSysMenu = FALSE;

    /* No scroll commands yet
    */
    iScrollMask = 0;
}
```

```
/*-----
FUNCTION _LOCAL BOOL ContextAdd(hwnd, conType)
DESCRIPTION Add an item of context info to the list.
```

Filling in the union feilds is up to the caller.

PARAMETERS HWND hwnd - Specifies handle to the window we are looking at.  
int conType -

RETURN TRUE if success

```
*/
LOCAL BOOL ContextAdd(HWND hwnd, int conType)
{
    CONTEXTITEM * pci;
    int c;

    if (pciLast != NULL)
    {
        /* Checksum the previous.
        */
        for (c = 0; c < sizeof(CONTEXTITEM); c++)
        {
            iCheckSum += ((PSTR) pciLast)[c];
        }
    }

    /* Must have a window ?
    */
    if (hwnd == NULL)
        return(FALSE);

    /* Allocate struct
    */
    pci = (CONTEXTITEM*) Nmalloc(sizeof(CONTEXTITEM));
    if (pci == NULL)
        return(FALSE);

    /* Set basic vars
    */
    pci->conType = conType;
    pci->iLevel = iGroupLevel;
    pci->hwnd = hwnd;

    /* Insert it after the pciLast.
    */
    if (pciFirst == NULL || pciLast == NULL)
    {
        /* At the start.
        */
        pci->pciNext = pciFirst;

        /* save top.
        */
        pciFirst = pci;
    }
    else
    {
        /* Insert after pciLast.
        */
    }
}
```

```

        pci->pciNext = pciLast->pciNext;

        /* Add to end.
        */
        pciLast->pciNext = pci;
    }

    /* The current pointer.
    */
    pciLast = pci;

    /* Return true so we continue enumerating.
    */
    return(TRUE);
}

/*-----
FUNCTION  _LOCAL BOOL HasKey(hMenu, iPos)
DESCRIPTION Check if the given menu has accelerator key.
             We check only \t, \a, or \b presents in the string

PARAMETERS HWND hMenu - Specifies handle to the given menu.
             int iPos - specifies posititon in the menu

RETURN

*/
_LOCAL BOOL HasKey(HMENU hMenu, int iPos)
{
    int i;

    if(! GetMenuString(hMenu, iPos, szCaptionBuf, sizeof(szCaptionBuf) - 1,
MF_BYPOSITION))
    {
        /* No text at all
        */
        return(FALSE);
    }

    for (i = 0; i < strlen(szCaptionBuf) - 1; i++)
    {
        if (szCaptionBuf[i] == '\t' || // For Windows Apps
            szCaptionBuf[i] == '\a' ||
            szCaptionBuf[i] == '\b') // For Microsoft Apps
        {
            /* Has TAB or ...
            */
            return(TRUE);
        }
    }
    return(FALSE);
}

```

```

/*-----
FUNCTION  _LOCAL void ContextAddAccel(HWND hwnd, HMENU hMenu)

DESCRIPTION Add the menu options to the context list.

PARAMETERS HWND hwnd - Specifies handle to the window we are looking at.
            HWND hMenu - Specifies handle to the given menu.

RETURN    None.
*/
_LOCAL void ContextAddAccel(HWND hwnd, HMENU hMenu)
{
    int iPos;
    int items;
    WORD State;

    if (hMenu == NULL)
    {
        /* No menu
        */
        return;
    }

    /* For all items
    */
    items = GetMenuItemCount(hMenu);
    for (iPos = 0; iPos < items; iPos++)
    {
        State = GetMenuState(hMenu, iPos, MF_BYPOSITION);
        if (State == -1)
            break;
        if (State & MF_POPUP)
        {
            /* Check submenu
            */
            ContextAddAccel(hwnd, GetSubMenu(hMenu, iPos));
        }
        else if (!(State & (MF_DISABLED | MF_GRAYED | MF_BITMAP |
MF_OWNERDRAW)))
        {
            if (HasKey(hMenu, iPos))
            {
                /* Add accelerator now
                */
                if (!ContextAdd(hwnd, CON_ACCEL))
                    return;
                pciLast->u.Acc.hMenu = hMenu;

                /* We use position as an ID
                */
                pciLast->u.Acc.id = GetMenuItemID(hMenu, iPos);
            }
        }
    }
}

```

```

    }

}

/*-----
FUNCTION  _LOCAL BOOL ContextAddMenu(HWND hwnd, HMENU hMenu)
DESCRIPTION Add the menu options to the context list.
PARAMETERS HWND hwnd - Specifies handle to the window we are looking at.
              HWND hMenu - Specifies handle to the given menu.
RETURN     None.
*/
_LOCAL BOOL ContextAddMenu(HWND hwnd, HMENU hMenu)
{
    int i;
    int items;
    WORD State;
    int numseparators;

    if (hMenu == NULL)
    {
        /* No menu
        */
        return(FALSE);
    }

    /* For all items
    */
    items = GetMenuItemCount(hMenu);
    numseparators = 0;
    for (i = 0; i < items; i++)
    {
        State = GetMenuState(hMenu, i, MF_BYPOSITION);
        if (State == -1)
            break;
        if (! ContextAdd(hwnd, (State & MF_POPUP) ? CON_MENUPOPUP :
CON_MENU))
            return(FALSE);

        /* Popups return different values
        */
        if (!(State & MF_POPUP))
        {
            /* Skip separator
            */
            if (State & MF_SEPARATOR)
                numseparators++;
        }

        if (pciLast->conType == CON_MENUPOPUP)
        {
            /* Store the entry number.

```

```

        */
        pciLast->u.MenuPop.iEntry = i;
        pciLast->u.MenuPop.iKeyPos = i - numseparators;
        pciLast->u.MenuPop.hMenu = hMenu;
    }
    else
    {
        /* Store ID.
        */
        pciLast->u.Menu.id = GetMenuItemID(hMenu, i);
        pciLast->u.Menu.hMenu = hMenu;
    }
}
return(TRUE);
}

```

```

/*-----
FUNCTION   void ContextNewLang(pLangEdit)

DESCRIPTION Change macro language.

PARAMETERS LPLANG pLangEdit - Specifies pointer to the new language.

RETURN     None.

```

```

*/
void ContextNewLang(LPLANG pLangEdit)
{
    char szFile[MAXFILENAME + 1];

    /* Destroy old language if present
    */
    LangChainDestroy(pLangCur);

    if (pLangEdit == NULL)
    {
        /* Try to open users language
        */
        Ini GetUserFile(szFile);
        lstrcat(szFile, ".LNG");
        pLangCur = LangLoad(szFile);
    }
    else
    {
        /* Try to copy from editor
        */
        pLangCur = LangChainMake(pLangEdit);
    }
    if (pLangCur == NULL)
    {
        /* Try to open default language
        */
        IniGetLangFile(szFile);
    }
}

```

T06050" 64025860

```

        pLangCur = LangLoad(szFile);
    }

}

/*-----
FUNCTION  _LOCAL LPLANG GetActiveLang()

DESCRIPTION A new task has been loaded so load new language
            or the default language.

PARAMETERS None.

RETURN    Pointer to the app specific language.
*/
_LOCAL LPLANG GetActiveLang()
{
    static HWND  hwndPrevActive = NULL;
    static LPLANG pActiveLang = NULL;
    HANDLE hTask;
    TASKENTRY te;
    char szFile[MAXFILENAME + 1];

    /* Active window changed
    */
    if (hwndPrevActive != hwndActive)
    {
        /* Save currently active window for the next call
        */
        hwndPrevActive = hwndActive;

        /* Get task handle
        */
        hTask = GetWindowTask(hwndActive);

        if (hTask == NULL)
        {
            /* No task ?!
            */
            pActiveLang = NULL;
        }
        else
        {
            /* Get module name
            */
            te.dwSize = (DWORD) sizeof(te);
            TaskFindHandle((TASKENTRY FAR *) &te, hTask);
            GetModuleFileName(te.hModule, (LPSTR)szFile, sizeof(szFile) - 1);

            /* Try to find language
            */
            for (pActiveLang = pLangCur->pNext; pActiveLang != NULL;
                pActiveLang = pActiveLang->pNext)

```



```

        {
            if (! lstrcmpi(szFile, pActiveLang->szFile))
            {
                /* Here it is
                */
                break;
            }
        }
    }

    /* Return pointer to the language or NULL
    */
    return(pActiveLang);
}

/*-----
FUNCTION  _LOCAL void AddLang(pLang, hwnd, szClass, szWndText, bMenuPopExist )
DESCRIPTION Add macro command from the language
PARAMETERS LPLANG pLang - Specifies pointer to the language.
            HWND hwnd - Specifies handle to the window we are looking at.
            PSTR szClass - Specifies pointer to the class name string.
            PSTR szWndText - Specifies pointer to the windows title.
            BOOL bMenuPopExist - TRUE if popup menu on the screen.
RETURN     None.
*/
_LOCAL void AddLang(LPLANG pLang, HWND hwnd, PSTR szClass, PSTR szWndText, BOOL
bMenuPopExist )
{
    LPGROUP pGroup;
    LPMACRO pMacro;
    HWND hwndMacro;

    if (pLang == NULL)
        /* No language selected
        */
        return;

    /* Try to find proper group
    */
    for (pGroup = pLang->pGroup; pGroup != NULL; pGroup = pGroup->pNext)
    {
        if (
            /* Default group
            */
            (pGroup->szClass == NULL && szClass == NULL)
            /* Class group
            */
            ||( pGroup->szClass != NULL && szClass != NULL && !
lstrcmp(pGroup->szClass, szClass)

```

```

szWndText))))      ) && (pGroup->szWndText == NULL || ! strcmp(pGroup->szWndText,
{
    /* Work with macros if the group has been found
    */
    for (pMacro = pGroup->pMacro; pMacro != NULL; pMacro = pMacro-
>pNext)
    {
        hwndMacro = hwnd;
        switch (pMacro->cmdType)
        {
            case CMD_WNDNAME:
            {
                /* Set alias name for the window
                */
                CONTEXTITEM * pci;
                char szBuf[MAXSTRING + 1];

                /* Look through the whole list
                */
                for (pci = pciFirst; pci != NULL; pci = pci-
>pciNext)
                {
                    /* We need CON_WIND or CON_ICON
                    */
                    if (pci->conType == CON_WIND
|| pci->conType == CON_ICON)
                    {
                        GetClassName(pci->hwnd,
szBuf, sizeof(szBuf)-1);

                        /* Compare class name and
                        */
                        if (pMacro->itemid ==
! strcmp(szBuf,
{
                    /* Window shouldn't
                    */
                    if (pci-
>u.Window.szName == NULL)
                    {
                        /* Set it
                        */
                        pci-
>u.Window.szName = pMacro->szName;
                        break;
                    }
                }
            }
        }
    }
    break;

```

```

    }

case CMD_MENU:
{
    /* Set alias name for the menu item
    */
    CONTEXTITEM * pci;

    for (pci = pciFirst; pci != NULL; pci = pci-
>pciNext)
    {
        /* We need CON_MENU with the same
        */
        if (pci->conType == CON_MENU
        &&
        pMacro->itemid == pci-
        >u.Menu.id)
        {
            /* Item shouldn't have alias yet
            */
            if (pci->u.Menu.szName ==
            NULL)
            {
                /* Set it
                */
                pci->u.Menu.szName =
                pMacro->szName;
                break;
            }
        }
        break;
    }
}

case CMD_MOUSE :
case CMD_JOURNAL :
{
    /* For mouse and journal macro we need to find
    */
    CONTEXTITEM * pci;
    char szBuf[MAXSTRING + 1];

    /* Class name of the window is the main
    */
    if (pMacro->szWndClass)
    {
        hwndMacro = NULL;

        /* Look through the whole list
        */

```

T06050" 64025850

```
pci->pciNext)
CON_WIND)
and child ID
>hwnd, szBuf, sizeof(szBuf)-1);
GetWindowWord(pci->hwnd, GWW_ID) &&
pMacro->szWndClass))
found it
pci->hwnd;

for (pci = pciFirst; pci != NULL; pci =
{
    if (pci->conType ==
    {
        /* Compare class name
        */
        GetClassName(pci-
        if (pMacro->itemid ==
            ! strcmp(szBuf,
        {
            /* we have
            */
            hwndMacro =
            break;
        }
    }
}
if (hwndMacro == NULL)
{
    /* No window
    */
    break;
}
}
default:
    if (bMenuPopExist)
        /* Can not do anything while popup
        */
        break;
    if (! ContextAdd(hwndMacro, CON_MACRO))
        /* Not enough memory
        */
        return;
    /* Add it
    */
    pciLast->u.pMacro = pMacro;
}
}
}
}
```

```

/*-----
FUNCTION  _LOCAL void AddLngCommands(hwnd, szClass, szWndText, bMenuPopExist )

DESCRIPTION Add macro command.

PARAMETERS HWND hwnd - Specifies handle to the window we are looking at.
           PSTR szClass - Specifies pointer to the class name string.
           PSTR szWndText - Specifies pointer to the windows title.
           BOOL bMenuPopExist - TRUE if popup menu on the screen.

RETURN    None.

*/
_LOCAL void AddLngCommands(HWND hwnd, PSTR szClass, PSTR szWndText, BOOL
bMenuPopExist )
{
    if (pLangCur == NULL)
    {
        /* No language at all
        */
        return;
    }

    /* Application specific  language
    */
    AddLang(GetActiveLang(), hwnd, szClass, szWndText, bMenuPopExist);

    /* Global language
    */
    AddLang(pLangCur, hwnd, szClass, szWndText, bMenuPopExist);
}

/*-----
FUNCTION  _LOCAL void AddScrollBarCommands(hwnd, ScrollMask, iCheckMask)

DESCRIPTION Create scroll bar command.

PARAMETERS HWND hwnd - Specifies handle to the window we are looking at.
           int ScrollMask - Specifies scroll mask.
           int iCheckMask - Specifies check mask.

RETURN    None.

*/
_LOCAL void AddScrollBarCommands(HWND hwnd, int ScrollMask, int iCheckMask)
{
    /* Scroll command with this name shouldn't be in the list twice
    */
    if (! (iScrollMask & iCheckMask))
    {
        /* This is first one

```

```

        */
        iScrollMask |= iCheckMask;

        if (! ContextAdd(hwnd, CON_SCROLL))
            /* Not enough memory
            */
            return;
        pciLast->u.ScrCom = SB_LINEUP | ScrollMask;

        if (! ContextAdd(hwnd, CON_SCROLL))
            /* Not enough memory
            */
            return;
        pciLast->u.ScrCom = SB_LINEDOWN | ScrollMask;

        if (! ContextAdd(hwnd, CON_SCROLL))
            /* Not enough memory
            */
            return;
        pciLast->u.ScrCom = SB_PAGEUP | ScrollMask;

        if (! ContextAdd(hwnd, CON_SCROLL))
            /* Not enough memory
            */
            return;
        pciLast->u.ScrCom = SB_PAGEDOWN | ScrollMask;

    }

}

/*-----
FUNCTION  _LOCAL void ContextAddScrollBars(hwnd, Style, cwc)
DESCRIPTION Add scroll bar commands.

PARAMETERS HWND hwnd - Specifies handle to the window we are looking at.
            LONG Style - Specifies windows style
            int cwc - Specifies window type.

RETURN    None.
*/
_LOCAL void ContextAddScrollBars(HWND hwnd, LONG Style, int cwc)
{
    switch (cwc)
    {
        case CWC_MDICLIENT:
            if (Style & WS_VSCROLL)
            {
                AddScrollBarCommands(hwnd, SCRLS_MDI, SCRLM_VMDI);
            }
            if (Style & WS_HSCROLL)
            {

```

```

        AddScrollBarCommands(hwnd, SCRLS_MDI | SCRLS_HORZ,
SCRLM_HMDI);
    }
    break;

    case CWC_SCROLLBAR :
        if (Style & SBS_VERT)
        {
            AddScrollBarCommands(hwnd, SCRLS_WIN, SCRLM_VERT);
        }
        else
        {
            AddScrollBarCommands(hwnd, SCRLS_WIN | SCRLS_HORZ,
SCRLM_HORZ);
        }
        break;

    default:
        if (Style & WS_VSCROLL)
        {
            AddScrollBarCommands(hwnd, 0, SCRLM_VERT);
        }
        if (Style & WS_HSCROLL)
        {
            AddScrollBarCommands(hwnd, SCRLS_HORZ,
SCRLM_HORZ);
        }
    }
}

```

```

/*-----
FUNCTION  _LOCAL void ContextAddWindSysCom(hwnd, Style)

DESCRIPTION Add system type commands for the window.

PARAMETERS HWND hwnd - Specifies handle to the given window.
            LONG Style - Specifies windows style

RETURN    None.

NOTE      Maximized MDI children are strange.
          The sys menu/restore is in the main menu of parent.
          They will not register normal WS_SYSMENU and restore boxes.
          Microsoft Excel violates even these rules !
          It will not set the WS_MAXIMIZE bit !

*/
_LOCAL void ContextAddWindSysCom(HWND hwnd, LONG Style)
{
    if (! (Style & WS_CHILD) || ! (Style & WS_MAXIMIZE))
    {
        /* Does the window have system command menu ?
        */
        if (! (Style & WS_SYSMENU))
            return;
    }
}

```

```

    }
    else
    {
        /* Can we get one ?
        */
        if (GetSystemMenu(hwnd, FALSE) == NULL)
            return;
    }

    /* Already got sysmenu type stuff ?
    */
    if (bChildSysMenu && (Style & WS_CHILD))
        return;
    bChildSysMenu = TRUE;

    /* Check to see if sys menu is already popped up.
    */
    if (hwndMenuSysPop == hwnd)
        /* Already popped.
        */
        return;

    /* Option to pull down the sys menu.
    */
    if (! ContextAdd(hwnd, CON_SYSCOM))
        return;
    /* The menu itself.
    */
    pciLast->u.SysCom = SC_KEYMENU;

    /* If the window is iconic then the others are not really available.
    ** Although they will say they are.
    */
    if (Style & WS_ICONIC)
        return;

    /* Option to close the window or app
    ** This is equiv. to double click on sys menu box.
    */
    if (! ContextAdd(hwnd, CON_SYSCOM))
        return;
    pciLast->u.SysCom = SC_CLOSE;

    /* Get the min/max controls seperatly for now.
    */
    if (Style & WS_MINIMIZEBOX)
    {
        if (! ContextAdd(hwnd, CON_SYSCOM))
            return;
        pciLast->u.SysCom = SC_MINIMIZE ;
    }

    /* If the window is maximized then we need a restore box.
    */
    if (Style & WS_MAXIMIZEBOX)
    {

```



```

        if (! ContextAdd(hwnd, CON_SYSCOM))
            return;
        pciLast->u.SysCom = (Style & WS_MAXIMIZE) ? SC_RESTORE :
SC_MAXIMIZE ;
    }
}

```

```

/*-----
FUNCTION  _LOCAL void ContextAddPMGroup(hwnd, Style)
DESCRIPTION Add content of Program Manager Group
PARAMETERS HWND hwnd - Specifies handle to the window we are looking at.
            LONG Style - Specifies windows style
RETURN    None.
*/

```

```

_LOCAL void ContextAddPMGroup(HWND hwnd, LONG Style)
{
    SHELLITEM si;
    BOOL  bRet;

    if (Style & WS_ICONIC)
    {
        /* We dont look inside iconic window, user cannot either
        */
        return ;
    }
    /* Window text is a group name
    */
    GetWindowText(hwnd, szCaptionBuf, sizeof(szCaptionBuf) - 1);

    /* Enumerate PM items inside the group
    */
    bRet = ShellGetFirstItem(&VCTalk, szCaptionBuf, &si);
    while (bRet)
    {
        /* We need command string to execute
        */
        if (si.szFile)
        {
            if (! ContextAdd(hwnd, CON_LAUNCH))
                /* not enough memory
                */
                return;

            /* Title is the name, file is the command string
            */
            pciLast->u.PMItem.szTitle = StringNearMake(si.szTitle);
            pciLast->u.PMItem.szFile = StringNearMake(si.szFile);
        }

        /* Next one ?
        */
    }
}

```

bRet = ShellGetNextItem(&VCTalk, &si);

}

}

/\*-----

FUNCTION \_LOCAL BOOL ContextAddWind(hwnd, checktype)

DESCRIPTION Check the window for useful context info.

PARAMETERS HWND hwnd - Specifies handle to the window we are looking at.  
int checktype - what type are we looking at. CW\_\*

RETURN TRUE if success.

NOTE Windows have the attributes of:

window handle  
window caption text, GetWindowText()  
parent handle, GetParent()  
rectangle. GetWindowRect() GetClientRect()  
child id number. GetDlgCtrlID()  
Enabled or disabled. IsWindowEnabled(hwnd)  
Active or Inactive. GetActiveWindow() ?  
Have focus ? GetFocus()

Window Class attributes. WNDCLASS. GetClassInfo  
style bit mask.  
class name. GetClassName ?  
module handle, Module name GetModuleFileName  
? cursor  
? icon  
? Menu bar resource name.

In the future we want to add special controls for known classes.

SCROLLBAR = bars may not be sub windows but part of the non client !

BUTTON = none needed but press.

STATIC = not needed but may label another control.

COMBOBOX = may have scrollbars, pull down, options inside ?

EDIT = scroll bars, new or dictated text ?

LISTBOX

We start from the bottom and work up. but previous parents are special.

Don't duplicate the parent of current focus.

\*/

\_LOCAL BOOL ContextAddWind(HWND hwnd, int checktype)

{

LONG Style;

int cwc;

int conType = CON\_WIND;

/\* default object type. \*/

char szClass[MAXSTRING + 1];

char szWndText[MAXSTRING + 1];

PREF\_FLAGS prefFlags = UserGetFlags();

```

if (hwnd == hwndPrvParent)
    /* We have already done with this window
    */
    return(TRUE);

/* Immediate children only.
*/
if ((checktype & CW_PARENTLEVEL) &&      ! (checktype & CW_HASFOCUS))
{
    if (hwndParent != GetParent(hwnd))
        /* Child of inactive window
        */
        return(TRUE);
}

/* Is the window iconized.
*/
Style = GetWindowLong(hwnd, GWL_STYLE);
if (Style & WS_ICONIC)
{
    conType = CON_ICON;
}

/* Is the window one of the known classes.
*/
GetClassName(hwnd, szClass, sizeof(szClass) - 1);

if (Style & WS_CHILD)
{
    /* check all control classes
    */
    for (cwc = 0; cwc < CWC_CHILD; cwc++)
    {
        if (! lstrcmpi(szClass, szPredefClass[cwc]))
            break;
    }
}
else
{
    /* It's popup
    */
    cwc = CWC_POPUP;
}

if (cwc == CWC_BUTTON && (Style & 0x0F) == BS_GROUPBOX)
{
    /* GroupBox is a special class
    */
    cwc = CWC_GROUPBOX;
}

/* Add children ScrollBars Control
*/
if ((prefFlags & PREF_Scroll) && cwc == CWC_SCROLLBAR)
{
    ContextAddScrollBars(hwnd, Style, cwc);
}

```

```

    }

    /* We must be focus or a parent of the focus to get menus and parts.
    */
    if ((checktype & CW_HASFOCUS) && (conType != CON_ICON))
    {

        /* Does the window have a menu bar ?
        */
        if (
            /* Not a child window.
            */
            ! (Style & WS_CHILD) &&
            /* Already have a menu, ONLY WANT ONE.
            */
            ! bMenuBarExist)
        {

            /* Get a menu bar if there is one.
            */
            if ((prefFlags & PREF_Menu) && ContextAddMenu(hwnd,
GetMenu(hwnd)))
            {
                bMenuBarExist = TRUE;
            }
        }

        /* FOR NOW, if a popup menu is active the window is not ???
        */
        if (! bMenuPopExist)
        {

            /* Add accelerators.
            */
            if (bMenuBarExist && (prefFlags & PREF_Accel))
            {
                ContextAddAccel(hwnd, GetMenu(hwnd));
            }

            /* Add contents of PMGroup
            */
            if (checktype == CW_HASFOCUS && cwc == CWC_PMGROUP &&
(prefFlags & PREF_WndChild))
            {
                ContextAddPMGroup(hwnd, Style);
            }

            /* Get system type commands.
            */
            if (prefFlags & PREF_SysCom)
            {
                ContextAddWindSysCom(hwnd, Style);
            }
        }
    }

```

```

        /* Add scroll commands
        */
        if (prefFlags & PREF_Scroll)
        {
            ContextAddScrollBars(hwnd, Style, cwc);
        }
    }

    /* Add macro commands
    */
    if (prefFlags & PREF_Macro)
    {

        /* Add non class specific macro commands only for the focus window
        */
        if (checktype == CW_HASFOCUS)
        {
            AddLngCommands(hwnd, NULL, NULL, bMenuPopExist);
        }

        /* Add windows specific macro commands for any active window
        */
        GetWindowText(hwnd, szWndText, sizeof(szWndText) - 1);
        AddLngCommands(hwnd, szClass, szWndText, bMenuPopExist);
    }
}

/* Add the window itself after its sub parts.
*/
if (! ContextAdd(hwnd, conType))
    return(FALSE);
pciLast->u.Window.cwc = cwc;

/* We need to add window even if a user doesn't want one
*/
if (! (checktype & CW_HASFOCUS) &&
    ((cwc == CWC_POPUP && ! (prefFlags & PREF_WndPopup)) ||
    (cwc != CWC_POPUP && ! (prefFlags & PREF_WndChild))))
{
    /* Not valid for phrase list
    */
    pciLast->u.Window.bForList = FALSE;
}
else
{
    /* Valid for phrase list
    */
    pciLast->u.Window.bForList = TRUE;
}

return(TRUE);
}

/*-----

```

```

FUNCTION  _LOCAL void ContextAddPopupMenu(void)
DESCRIPTION Get a popped up or selected menu or menu tree.
PARAMETERS None.
RETURN    None.
*/
_LOCAL void ContextAddPopupMenu(void)
{
    HMENU hMenu;
    LONG Style;
    HWND hwnd = NULL;
    int iLevel = 0;

    /* Start
    */
    bMenuPopExist = FALSE;

    if (HookGet_MenuLevel() == -1)
    {
        /* No menu at all
        */
        return;
    }

    while (1)
    {
        /* Is there a menu popped up.
        */
        hMenu = HookGet_Menu(iLevel ++);
        if (hMenu == NULL)
        {
            /* No menu at all
            */
            return;

            /* Get menu from its owner window.
            ** Do just once.
            */
            if (hwnd == NULL)
            {
                bMenuPopExist = TRUE;
                hwnd = HookGet_MenuWnd();
                if (GetWindowTask(hwnd) == GetCurrentTask()) {
                    /* Don't look at Voice control
                    */
                    return;
                }
                Style = GetWindowLong(hwnd, GWL_STYLE);
            }

            /* If the popup menu is part of the main menu bar,
            ** then mark that we already have it.
            ** NOTE:

```

```

    /** GetMenu() is undefined for WS_CHILD types.
    */
    if (! (Style & WS_CHILD))
    {
        if (hMenu == GetMenu(hwnd))
            bMenuBarExist = TRUE;
    }

    /* Add menu without accelerators
    */
    if (UserGetFlags() & PREF_Menu)
    {
        if (ContextAddMenu(hwnd, hMenu))
        {
            iGroupLevel++;
        }
    }

    /* Is it a system menu
    */
    if (hMenu == GetSystemMenu(hwnd, FALSE))
    {
        hwndMenuSysPop = hwnd;
    }
}

/*-----
FUNCTION  BOOL CALLBACK ContextEnumProc(hwnd, lParam)
DESCRIPTION Callback function that receives window handles as
a result of a call to the EnumWindows function.
PARAMETERS HWND hwnd - Specifies handle of the target window.
LONG lParam - What do we do with the data once we have it ?
RETURN    Return nonzero to continue enumeration.
*/
BOOL FAR PASCAL ContextEnumProc(HWND hwnd, LONG lParam)
{
    return (ContextAddWind(hwnd, (int) lParam));
}

/*-----
FUNCTION  _LOCAL char StringGetSysChar(String)
DESCRIPTION Get underlined symbol from the menu item.
PARAMETERS PSTR String - Specifies menu string.
RETURN    Underlined symbol.
*/

```

\_LOCAL char StringGetSysChar(PSTR String)

```
{
    while (*String)
    {
        if (*(String++) == '&')
        {
            /* We have found &
            */
            break;
        }
    }
    /* Return address of the next one
    */
    return(*String);
}
```

-----

FUNCTION \_LOCAL int ContextPakWind(hwnd)

DESCRIPTION Pak a string description for the window type object.  
User pciLast to identify the object.

PARAMETERS HWND hwnd - Specifies handle to the window we are looking at.

RETURN Length of the caption text.

\*/

\_LOCAL int ContextPakWind(HWND hwnd)

```
{
    int len;

    /* If window not active then ignore it.
    */
    if (
        (! IsWindowEnabled(hwnd)
        || ! IsWindowVisible(hwnd))) /* Not really working ??? */
        return(0);

    /* What is its caption text ?
    */
    len = GetWindowText(hwnd, szCaptionBuf, sizeof(szCaptionBuf) - 1);

    /*
    ** What is its class.
    */
    switch (pciLast->u.Window.cwc)
    {
        case CWC_EDIT:
        case CWC_COMBOBOX:
        case CWC_LISTBOX:
        case CWC_SCROLLBAR:
            /* Edit/Comb/List captions are the current text inside them ?
            */
            len = 0;
            break;
    }
}
```



```

case CWC_GROUPBOX:
case CWC_STATIC:
    /* If static or group box has & it lable something
    */
    if (! StringGetSysChar(szCaptionBuf))
    {
        len = 0;
    }
    break;

default:
    ;

```

```

    }
    return(len);
}

```

```

/*-----
FUNCTION  _LOCAL int ContextPakMenu(hMenu, idItem, fuFlags)
DESCRIPTION Get an option from a menu.
PARAMETERS HMENU hMenu - Specifies handle to the menu.
            int idItem - Specifies item ID.
            UINT fuFlags - Specifies item flags.
RETURN     Length of the caption text.
NOTE       When sys menus of child windows are popped up:
            they have a popup menu type with a caption of junk ?
            The high MF_ values str not valid for MF_POPUP or menu bars.
            high = the number of entries in the popup.
*/
_LOCAL int ContextPakMenu(HMENU hMenu, int idItem, UINT fuFlags)
{

```

```

    WORD State;
    int len = 0;

```

```

    if (hMenu == NULL) return(0);

```

```

    State = GetMenuState(hMenu, idItem, fuFlags);
    if (State == -1) return(0);

```

```

    /* Is the item available grayed, disabled ?
    ** -1 == not exist.
    */

```

```

    if ((State & MF_DISABLED )
        ||(State & MF_GRAYED ))
        return 0;

```

```

    if (! (State & MF_POPUP))
    {
        if ((State & MF_BITMAP)

```

```

        || (State & MF_OWNERDRAW))
        return 0;;
    }

    /* Get the text description.
    */
    len = GetMenuString(hMenu, idItem, szCaptionBuf, sizeof(szCaptionBuf) - 1, fuFlags);

    return(len);
}

```

```

/*-----
FUNCTION  _LOCAL int ContextPakSysCom(hwnd, iSysCom)
DESCRIPTION Create system command string.
PARAMETERS HWND hwnd - Specifies handle to the window we are looking at.
            int iSysCom - SC_...
RETURN    Length of the caption text.
*/
_LOCAL int ContextPakSysCom(HWND hwnd, int iSysCom)
{
    char Str[MAXSTRING + 1];
    int len = 0;

    switch (iSysCom)
    {
        case SC_KEYMENU:
        case SC_MOUSEMENU:
            /* We can get other options by pulling down the sys menu.
            */
            len = wsprintf(
                szCaptionBuf,
                "%s %s",
                (LPSTR)UserGetDefWord((GetWindowLong(hwnd, GWL_STYL
E) & WS_CHILD) ? IDW_CHILD : IDW_POPUP),
                (LPSTR)UserGetDefWord(IDW_SYSMENU));
            break;

        case SC_CLOSE:           /* May be close window or app. */
        case SC_MINIMIZE:
        case SC_MAXIMIZE:
        case SC_RESTORE:
            /* List these visible controls seperately.
            */
        default:
            GetMenuString(GetSystemMenu(hwnd, FALSE), iSysCom, Str,
MAXSTRING, MF_BYCOMMAND);
            len = StringClip(Str);
            if (len)
            {
                len = wsprintf(
                    szCaptionBuf,

```

```

        "%s %s",
        (LPSTR)Str,
        (LPSTR)UserGetDefWord((GetWindowLong(hwnd,GWL
_STYLE) & WS_CHILD) ? IDW_CHILD : IDW_POPUP));
    }

    return(len);
}

```

```

/*-----
FUNCTION  _LOCAL int ContextPakScroll(iScrCom)
DESCRIPTION Create scroll command string.
PARAMETERS int iScrCom - Specifies scroll command.
RETURN     Length of the caption text.
*/
_LOCAL int ContextPakScroll(int iScrCom)
{

```

```

    int len;
    int idWord;

    /* First try all type of horizontall scroll
    */
    if (iScrCom & SCRLS_HORZ)
    {
        switch (pciLast->u.ScrCom & SCRLS_ACT)
        {
            case SB_LINEUP:
                /* line left
                */
                idWord = IDW_LINELEFT;
                break;
            case SB_LINEDOWN:
                /* line right
                */
                idWord = IDW_LINERIGHT;
                break;
            case SB_PAGEUP:
                /* page left
                */
                idWord = IDW_PAGELEFT;
                break;
            case SB_PAGEDOWN:
                /* page right
                */
                idWord = IDW_PAGERIGHT;
                break;
        }
    }

    /* Now all type of vertical scroll

```

```

|
*/
_LOCAL int ContextPak(void)
{
    int len;
    HWND hwnd = pciLast->hwnd;

    *szCaptionBuf = '\0';

    switch (pciLast->conType)
    {
        case CON_WIND:
        case CON_ICON:
            /* Does the user want to have window names ?
            */
            if (! pciLast->u.Window.bForList)
            {
                len = NULL;
                break;
            }
            /* Does alias name exist ?
            */
            if (pciLast->u.Window.szName)
            {
                lstrcpy(szCaptionBuf, pciLast->u.Window.szName);
                len = lstrlen(szCaptionBuf);
            }
            /* Try to get caption
            */
            else
            {
                len = ContextPakWind(hwnd);
            }
            break;

        case CON_SYSCOM:
            /* The system command for the window.
            */
            len = ContextPakSysCom(hwnd, pciLast->u.SysCom);
            break;

        case CON_SCROLL:
            len = ContextPakScroll(pciLast->u.ScriCom);
            break;

        case CON_MENU:
            /* Does alias name exist ?
            */
            if (pciLast->u.Menu.szName)
            {
                lstrcpy(szCaptionBuf, pciLast->u.Menu.szName);
                len = lstrlen(szCaptionBuf);
            }
            /* Get an item from a popped up menu.
            */

```

```

        else {
            len = ContextPakMenu(pciLast->u.Menu.hMenu, pciLast-
>u.Menu.id, MF_BYCOMMAND);
        }
        break;

    case CON_MENUPOPUP:
        /* Read an item from the menu bar.
        */
        len = ContextPakMenu(pciLast->u.MenuPop.hMenu, pciLast-
>u.MenuPop.iEntry, MF_BYPOSITION);
        break;

    case CON_ACCEL:
        /* Accelerator has the same text as a menu item (it available thought)
        */
        len = GetMenuString(pciLast->u.Acc.hMenu, pciLast->u.Acc.id,
szCaptionBuf,
                            sizeof(szCaptionBuf) - 1, MF_BYCOMMAND);
        break;

    case CON_LAUNCH :
        /* PM item title
        */
        lstrcpy(szCaptionBuf, pciLast->u.PMItem.szTitle);
        len = lstrlen(szCaptionBuf);
        break;

    case CON_MACRO:
        /* Macro name
        */
        lstrcpy(szCaptionBuf, (pciLast->u.pMacro)->szName);
        len = lstrlen(szCaptionBuf);
        break;

    default: return(0);
}

/* Chop out the ampersands (&) and tabs.
*/
if (len)
    len = StringClip(szCaptionBuf);
else
    *szCaptionBuf = '\0';

if (len > iCaptionLen)
    iCaptionLen = len;

#ifdef DEBUG_DLG
/* Pack debug info
*/
if (DebugFlag & DEBUG_ContFull)
{
    len = ContextPakDebug();
    if (len > iCaptionLen) iCaptionLen = len;
}

```

#endif

```
/* Return length of the string
*/
return(len);
```

}

```
/*-----
FUNCTION  BOOL ContextCheck(bPrefChange)
DESCRIPTION Hook the context window to the status window.
PARAMETERS  BOOL bPrefChange - Rebuild list anyway
RETURN      TRUE = A change in the context ?
NOTE        This is called every so often to check for context changes.
              Watch for the change in focus thru the hook routines ?
              Menus don't change the focus ! we must watch messages for them !
              When we select an icon the focus = null the active window is icon.
```

```
*/
```

```
BOOL ContextCheck(BOOL bPrefChange)
```

```
{
```

```
    int    checktype;
    int    changetype;
    unsigned PrevCheckSum;
```

```
    changetype = HookGet_Change();
```

```
    /* Does anything change ?
```

```
    */
```

```
    if (changetype == HCHANGE_NONE && ! bPrefChange)
        return(FALSE);
```

```
    /* Set up to enumerate the windows.
```

```
    */
```

```
    if (lpprocContext == NULL)
```

```
    {
```

```
        lpprocContext = MakeProcInstance(ContextEnumProc, VChInst);
```

```
    }
```

```
    /* First we check context save options (when old focus valid).
```

```
    */
```

```
    if (GetWindowTask(GetActiveWindow()) == GetCurrentTask())
```

```
    {
```

```
        if (IsWindow(hwndFocus) && (! IsIconic(hwndActive) || hwndActive ==
hwndFocus))
```

```
        {
```

```
            /* Context still good for now, but we need to check preferences
```

```
            */
```

```

        if (! bPrefChange)
        {
            return(FALSE);
        }
    }
    else
    {
        /* We cannot find our active window.
        ** Don't look to it.
        */
        hwndFocus = 0;
    }
}
else
{
    /* Who is active now.
    */
    hwndActive = GetActiveWindow();

    /* Who has focus right now.
    */
    hwndFocus = GetFocus();

    /* We should start
    */
    if (! hwndFocus)
        hwndFocus = hwndActive;
}

/*
** restart the context list.
*/
PrevCheckSum = iCheckSum;    /* Save the previous to compare. */
ContextListInit();

/*
** Check for a pop up menu active.
** ALWAYS highest focus priority.
*/
ContextAddPopupMenu();

if (hwndFocus)
{
    /*
    ** Get those windows that are children of the current focus.
    ** NOTE: Items in the immediate focus should be on top !
    ** Move up the hierarchy to the modal level or the non WS_CHILD ?
    */

    hwndParent = hwndFocus;
    hwndPrvParent = NULL;
    checktype = 0;

    while (hwndParent != NULL)

```

```

    {
        if (! IsWindowEnabled(hwndParent)) /* The previous was top. */
            break;

        if (! IsIconic(hwndParent))
        {
            EnumChildWindows(hwndParent, lpprocContext, 'checktype);
            iGroupLevel ++;
        }

        /*
        ** Store the parent level. (May not be a real option.)
        */
        ContextAddWind(hwndParent, CW_HASFOCUS | checktype);
        hwndPrvParent = hwndParent; /* Don't duplicate in siblings. */
        iGroupLevel ++;
        checktype = CW_PARENTLEVEL;

        /*
        ** Break after Active window
        */
        if (hwndParent == hwndActive)
        {
            break;
        }

        /*
        ** Does it have a parent ?
        */
        hwndParent = GetParent(hwndParent);
    }

}

/*
** Get other applications. except if someone above is system modal.
** WS_OVERLAPPED and WS_POPUP type windows.
*/
EnumWindows(lpprocContext, 0);

ContextAdd(NULL, 0); /* Checksum the last. */

return(PrevCheckSum != iCheckSum || changetype > HCHANGE_POSSIBLE);
}

```

```

/*-----
| FUNCTION void ContextListAdd(void)
| DESCRIPTION Build a list of siblings and children.
| PARAMETERS None.
| RETURN None.
|

```



```

*/
void ContextListAdd(void)
{
    int len;
    int iEntry = 0;

    ContextCheck(FALSE);          /* One final check before packing. */

    iCaptionLen = 13;              /* Minimum size. */

#ifdef DEBUG_DLG
    iDebugCapLen = 0;
#endif

    for (pciLast = pciFirst; pciLast != NULL; pciLast = pciLast->pciNext, iEntry++)
    {
        len = ContextPak();
        if (!len) continue;

        /* Send a message adding the window caption to the list
        ** in the dialog.
        */
        if (! PhraseListAdd(szCaptionBuf, iEntry)) break;
    }

#ifdef DEBUG_DLG
    /* Set the tabs and columns.
    */
    if (DebugFlag & DEBUG_ContFull)
    {
        ContextTabs[0] = (iCaptionLen + 4) * 10;
        ContextTabs[1] = (iCaptionLen + 12) * 10;
        ContextTabs[2] = iCaptionLen + 16 + iDebugCapLen;
    }
#endif
}

/*-----
| FUNCTION void ContextListSelect(iEntry)
| DESCRIPTION The user selected a word from the list.
|              Take some default MACRO action based on the context type
| PARAMETERS int iEntry - Specifies numer of list item;
| RETURN None.
|
| */
void ContextListSelect(int iEntry)
{
    HWND hwnd;
    MACRO macro;

```

```

if (iEntry < 0) return;

/*
** Find the window in the list.
*/
for (pciLast = pciFirst; iEntry; iEntry --)
{
    if (pciLast == NULL)
        return; /* THIS SHOULD NEVER HAPPEN */
    pciLast = pciLast->pciNext;
}
hwnd = pciLast->hwnd;

/* We keep focus and it valid.
*/
if (GetWindowTask(GetActiveWindow()) == GetCurrentTask())
{
    SetFocus(hwndFocus);
}

/* Default macros are to be executed on hwnd.
*/
macro.szWndClass = NULL;
macro.szDesc     = NULL;
macro.pNext      = NULL;

switch (pciLast->conType) {

    case CON_SYSCOM:
        /* A system command from the system command menu to the window.
        ** PostMessage(hwnd, WM_SYSCOMMAND, iEntry, NULL);
        */
        macro.cmdType      = CMD_SYSTEM;
        macro.Cmd.System.wCmd = pciLast->u.SysCom;
        break;

    case CON_SCROLL:
        /* PostMessage
        */
        macro.cmdType      = CMD_MESSAGE;
        macro.Cmd.Msg.wMsg = (pciLast->u.ScrCom & SCRLS_HORZ) ?
WM_HSCROLL : WM_VSCROLL;
        macro.Cmd.Msg.wParam = pciLast->u.ScrCom & SCRLS_ACT;

        if (pciLast->u.ScrCom & SCRLS_WIN)
        {
            macro.Cmd.Msg.lParam = MAKELONG(0, hwnd);
            hwnd = GetParent(hwnd);
        }
        else
        {
            macro.Cmd.Msg.lParam = 0L;
        }
        break;

```

```

case CON_ICON:
    /* Restore the iconic window.
    ** NOTE:
    ** Iconic windows don't get focus. they just activate.
    ** OpenIcon(hwnd);
    */
    macro.cmdType = CMD_SYSTEM;
    macro.Cmd.System.wCmd = SC_RESTORE;
    break;

case CON_WIND:
    if ((pciLast->u.Window.cwc == CWC_STATIC) || (pciLast-
>u.Window.cwc == CWC_GROUPBOX))
    {
        GetWindowText(hwnd, szCaptionBuf, sizeof(szCaptionBuf) - 1);
        macro.cmdType = CMD_KEY;
        macro.Cmd.Key.cKey = (char)
VkKeyScan(StringGetSysChar(szCaptionBuf));
        macro.Cmd.Key.AltPressed = (BYTE) 1;
        macro.Cmd.Key.ShiftPressed = (BYTE) 0;
        macro.Cmd.Key.CtrlPressed = (BYTE) 0;
    }
    else
    {
        /* Choose the window as the current window. For top level
        windows this
        ** will result in their being activated. For items in dialog boxes
        ** this will result in their being selected.
        */
        macro.cmdType = CMD_SELECT;
    }
    break;

case CON_MENUPOPUP:
    /* An item on the windows menu bar.
    ** Pull down the popup menu.
    */
    macro.cmdType = CMD_MENUPOPUP;
    macro.Cmd.MenuPopup.iKeyPos = pciLast->u.MenuPop.iKeyPos;

    if (GetMenu(hwnd) == pciLast->u.MenuPop.hMenu)
        macro.Cmd.MenuPopup.wLevel = 0;
    else
        macro.Cmd.MenuPopup.wLevel = 1;
    break;

case CON_MENU:
    /* A menu item in the active menu.
    ** Execute the menu item.
    ** PostMessage(hwnd, WM_COMMAND, iEntry, NULL);
    */

```

```

if (hwndMenuSysPop)
{
    /* Menu item chosen from system menu.
    */
    macro.cmdType = CMD_SYSTEM;
    macro.Cmd.System.wCmd = pciLast->u.Menu.id;
}
else
{
    /* Menu item chosen from the menu bar.
    */
    macro.cmdType = CMD_MENU;
    macro.Cmd.Menu.id = pciLast->u.Menu.id;
}
break;

```

```

case CON_ACCEL:
    /* Accelerator key
    */
    macro.cmdType = CMD_MENU;
    macro.Cmd.Menu.id = pciLast->u.Acc.id;
    break;

```

```

case CON_LAUNCH:
    /* Just execute
    */
    macro.cmdType = CMD_LAUNCH;
    macro.szDesc = pciLast->u.PMItem.szFile;
    break;

```

```

case CON_MACRO:
    macro.cmdType = pciLast->u.pMacro->cmdType;
    macro.Cmd = pciLast->u.pMacro->Cmd;
    macro.itemid = pciLast->u.pMacro->itemid;
    macro.szDesc = pciLast->u.pMacro->szDesc;
    break;

```

```

default :
    return;

```

```

}

```

```

VCM_Execute(&macro, hwnd);

```

```

}

```

```

/*
** File: HOOK.C
**
** Module for Hooking Window's queue and tracking relevant messages.
**
** Interface functions: HookGet_Change
**                      HookGet_Menu
**                      HookGet_MenuAtLevel
**                      HookGet_MenuLevel
**                      HookGet_MenuWnd
**                      HookInstall
**                      HookJournalBusy
**                      HookFreeJournal
**                      Record
**
** Exported functions: HookMain
**                      HookGetMsgProc
**                      HookSndMsgProc
**                      PlayProc
**                      RecProc
**
** Private functions: HookMenuClear
**                      HookMessage
**                      PlayNotify
**                      RecNotify
**
*****
*****/

#include <windows.h>
#include "vtools.h"

typedef struct
{
    // Another message type
    DWORD lParam;           /* This was backwards before ? */
    WORD wParam;
    WORD wMsg;
    HWND hWnd;
} CALLWNDPROC;             /* NOTE: Parameters are oposite of LPMMSG ? */

typedef CALLWNDPROC FAR *LPCALLWNDPROC;

/*-----
|
| Module local variables.
|
*/
HANDLE hInst;              // Instance Handle given in LibMain()
HHOOK hGetMsgHook;         // Handle to the getmessage hook
HHOOK hSndMsgHook;         // Handle to the callwndproc hook
HHOOK hJournalHook;        // Current journal record/playback hook function

/*-----
|
| --- Variables for Playback ---
|

```

```

*/
static LPRECORD lpJmlList; // Handle to the list of journal events
static BOOL bJournalBusy; // Is the DLL busy recording or playing back?
static DWORD dwInitPlaybackTime; // Initial time of Playback() call
static short sPlaybackSpeed; // Speed given to Playback() (0 or -1)
static DWORD dwPrevMsgTime; // Time of previously played back event

static HWND hWndNotify;
static UINT wMsgNotify;
static UINT wStopKey;
static UINT wMouRec;

/*-----
| --- Context manager tracking. ---
|
*/
static int Hook_Change; // context change type. */
static HWND Hook_MenuhWnd; // The window owning the menu. */
static int Hook_MenuLevel; // The menu stack level. -1=none */
static HMENU Hook_MenuSelect; // Selected item from the current level. */

static enum
{
    /*
    ** If we are tracking a multi message operation.
    */
    HT_NONE, // Watch for nothing. */
    HT_ACCEL, // Watch for an accelerator key press. */
} Hook_Track;

#define MENUSTACKQTY 6 // How many sub levels to store. */

static HMENU Hook_MenuStack[MENUSTACKQTY]; // currently active menu. */

/*-----
|
| FUNCTION int CALLBACK HookMain(hinst, wDataSeg, wHeapSize, lpszCmdLine)
|
| DESCRIPTION Part of the LibMain that belongs to the hook system.
|
| PARAMETERS HINSTANCE hinst - Identifies the instance of the DLL.
| WORD wDataSeg - Specifies the value of the data
| segment (DS) register.
| WORD wHeapSize - Specifies the size of the heap defined
| in the module-definition file.
| LPSTR lpszCmdLine - Points to a null-terminated string
| specifying command-line information.
|
| RETURN 1 if it is successful. Otherwise, it should return 0.
|
*/
int CALLBACK HookMain(HINSTANCE hinst, WORD wDataSeg, WORD wHeapSize, LPSTR
lpszCmdLine)
{

```

```

hInst = hinst;
bJournalBusy = FALSE;
hGetMsgHook = NULL;
hSndMsgHook = NULL;

Hook_Change = HCHANGE_NONE;
Hook_MenuLevel = -1;
Hook_Track = HT_NONE;

return (TRUE);
}

```

```

/*-----
FUNCTION   int WINAPI HookGet_Change(void)
DESCRIPTION Has part of the context changed.
            Because looking for changes is not an exact science we know some
            events are always a change and some are just possible.
            Keep 2 flags.
PARAMETERS None.
RETURN     Hook change status.
*/
int WINAPI HookGet_Change(void)
{
    int Prev;

    Prev = Hook_Change;
    Hook_Change = HCHANGE_NONE;

    return(Prev);
}

```

```

/*-----
FUNCTION   HMENU WINAPI HookGet_Menu(level)
DESCRIPTION Return the handle to the current popped up menu.
PARAMETERS int level - the inverse of the menu stack level. 0=top-most
RETURN     NULL = no menu is popped up
*/
HMENU WINAPI HookGet_Menu(int level)
{
    if (level > Hook_MenuLevel) return(NULL);

    return(Hook_MenuStack[Hook_MenuLevel - level]);
}
/*-----

```

```

| FUNCTION  HMENU WINAPI HookGet_MenuAtLevel(level)
|
| DESCRIPTION Return the handle to the menu at the given level.
|
| PARAMETERS  int level - the menu stack level. 0=top-most
|
| RETURN      NULL = no menu is popped up.

```

```

| */
| HMENU WINAPI HookGet_MenuAtLevel(int level)
| {
|     if (level > Hook_MenuLevel) return(NULL);
|
|     return(Hook_MenuStack[level]);
| }

```

```

| /*-----
|
| FUNCTION  int WINAPI HookGet_MenuLevel()
|
| DESCRIPTION Return the menu level.
|
| PARAMETERS  None.
|
| RETURN      The menu level : NULL = no menu is popped up.

```

```

| */
| int WINAPI HookGet_MenuLevel()
| {
|     return(Hook_MenuLevel);
| }

```

```

| /*-----
|
| FUNCTION  HWND WINAPI HookGet_MenuWnd(void)
|
| DESCRIPTION Returns the owner of the popped up window.
|             Only valid if there IS a popped up menu !
|
| PARAMETERS  None.
|
| RETURN      Handle to the window.

```

```

| */
| HWND WINAPI HookGet_MenuWnd(void)
| {
|     return(Hook_MenuhWnd);
| }

```

```

| /*-----
|
| FUNCTION  static void HookMenuClear(void)
|
| DESCRIPTION Clear menu toggles.

```



PARAMETERS None.

RETURN None.

\*/

static void HookMenuClear(void)

{

if (Hook\_MenuLevel == -1) return;

Hook\_MenuLevel = -1;

/\* No popup menu. \*/

Hook\_Change |= HCHANGE\_DEFINATE;

}

/\*-----

FUNCTION static void PASCAL HookMessage(hWnd, wParam, lParam)

DESCRIPTION Check for common context indication messages.

Use command message checker for PostMessage and SendMessage  
because we never really know which will be used.

PARAMETERS HWND hWnd - Specifies the handle of the window

UINT wParam - Specifies the message

WORD lParam - Specifies 16 bits of additional  
message-dependent information

LONG lParam - Specifies 16 bits of additional  
message-dependent information

RETURN None.

\*/

static void PASCAL HookMessage(HWND hWnd, UINT wParam, WORD lParam, LONG lParam)

{

switch (wParam)

{

/\*

\*\* Menu level tracking.

\*/

case WM\_INITMENU:

/\*

\*\* The bottom level menu is initialized.

\*/

Hook\_MenuhWnd = hWnd;

Hook\_MenuLevel = -1;

Hook\_MenuSelect = NULL;

Hook\_Track = HT\_NONE;

Hook\_Change |= HCHANGE\_DEFINATE;

break;

case WM\_INITMENUPOPUP:

/\*

\*\* The menu will pop up onto the screen.

\*\* NOTE: The context manager needs this to tell if a menu is up.

```

    */
    if (Hook_MenuSelect == wParam)
    {
        if (Hook_MenuLevel >= MENUSTACKQTY-1) break;      /*
SORRY */
        Hook_MenuLevel ++;
    }
    else
    {
        /*
        ** NOTE:
        ** Of the Popup is initialized without having selected it
        ** then it is not a normal menu popup ? What do i do ?
        ** NOTE:
        ** This works for custom popups.
        */
        Hook_MenuLevel = 0;      /* Don't know where this is from ?

        Hook_Track = HT_ACCEL;
    }

    Hook_MenuSelect = NULL;
    Hook_MenuStack[Hook_MenuLevel] = wParam;
    Hook_Change |= HCHANGE_DEFINATE;
    break;

case WM_MENUSELECT:
    /*
    ** Watch for the pop up menu being removed.
    ** or the select being moved.
    ** wParam = the item seelcted, (handle if popup)
    ** HIWORD(IParam) = our parent.
    */

    if (wParam == 0 && IParam == 0xFFFFL)
    {
        HookMenuClear();
        break;
    }
    if (Hook_MenuLevel == -1)
    {
        Hook_MenuStack[++ Hook_MenuLevel] = HIWORD(IParam);
        Hook_Change |= HCHANGE_DEFINATE;
    }
    else
    {
        if (HIWORD(IParam) == Hook_MenuSelect)
        {
            /*
            ** NOTE:
            ** This occurs if the menu select is moved back to the
parent-

            ** But the child is left on the screen ?
            */
            Hook_MenuLevel ++;      /* same as
last. */

```

```

        Hook_Change |= HCHANGE_DEFINATE;
    }
    else
    {
        while (Hook_MenuLevel > 0)
        {
            if (HIWORD(lParam) ==
Hook_MenuStack[Hook_MenuLevel])
                break;
            Hook_MenuLevel--;
            Hook_Change |= HCHANGE_DEFINATE;
        }
    }

    Hook_Track = HT_NONE;
    Hook_MenuSelect = wParam;
    break;

```

case WM\_SYSCOMMAND:

```

/*
** Check for the window being maximized, minimized or restored.
*/
switch (wParam)
{
    case SC_MAXIMIZE :
    case SC_MINIMIZE :
    case SC_RESTORE :
        Hook_Change |= HCHANGE_DEFINATE;
        break;
}

```

case WM\_COMMAND:

```

/*
** Clear the menu if present.
** NOTE: Accelerator keys only exit with a WM_COMMAND
*/
if (Hook_Track == HT_ACCEL)
    HookMenuClear();
break;

```

case WM\_ACTIVATEAPP:

```

/*
** We are changing applications.
*/
Hook_Change |= HCHANGE_TASK;
break;

```

case WM\_ACTIVATE:

```

/*
** The window activation is changing. similar to focus.
*/

```

case WM\_SETFOCUS:

case WM\_KILLFOCUS:

```

/*

```

```

    ** The focus is changing.
    */
    Hook_Change |= HCHANGE_POSSIBLE;
    break;

```

```

case WM_SETTEXT:
    /*
    ** Some text is being set to a window or control.
    ** Most likely it is a change.
    */
    Hook_Change |= HCHANGE_DEFINITE;
    break;

```

```

case WM_SHOWWINDOW:
    Hook_Change |= HCHANGE_DEFINITE;
    break;

```

```

case WM_CREATE:
    /*
    ** The window is created.
    */

```

```

case WM_PAINT:
case WM_NCPAINT:
case WM_NCCALCSIZE:
case WM_CTLCOLOR:
case WM_ENTERIDLE:
    /*
    ** NOTE: It could be (Not necessary) a change.
    */
    Hook_Change |= HCHANGE_POSSIBLE;
    break;

```

```

    }

```

```

}

```

```

/*-----

```

**FUNCTION** DWORD CALLBACK HookGetMsgProc(nCode, wParam, lParam)

**DESCRIPTION** The HookGetMsgProc function is a callback function that the system calls whenever the GetMessage function has retrieved a message from an application queue. The system passes the retrieved message to the callback function before passing the message to the destination window procedure.

**PARAMETERS** int nCode - Specifies whether the callback function should process the message or call the CallNextHookEx function. If this parameter is less than zero, the callback function should pass the message to CallNextHookEx without further processing.

WORD wParam - Specifies a NULL value.

LPMMSG lParam - Points to an MSG structure that contains information about the message.

```

| RETURN    The callback function should return zero.
|
| */
DWORD CALLBACK HookGetMsgProc(int nCode, WORD wParam, LPMSG lpMsg)
{
    if (nCode == HC_ACTION)
    {
        HookMessage(lpMsg->hwnd, lpMsg->message, lpMsg->wParam, lpMsg->
        lParam);
        if (lpMsg->message == WM_MOUSEMOVE)
        {
            lpMsg->wParam &= ~MK_MBUTTON;
        }
    }

    return CallNextHookEx(hGetMsgHook, nCode, wParam, (LONG)lpMsg);
}

```

```

/*-----
FUNCTION  DWORD CALLBACK HookSndMsgProc(nCode, wParam, lpMsg)

DESCRIPTION Hooks all SendMessage calls.

PARAMETERS int nCode      -Specifies whether the callback function
                           should process the message or call the
                           CallNextHookEx function. If this parameter
                           is less than zero, the callback function
                           should pass the message to CallNextHookEx
                           without further processing.
                           WORD wParam      -Specifies whether the message is sent by
                           the current task. This parameter is
                           nonzero if the message is sent;
                           otherwise, it is NULL.
                           LPCALLWNDPROC lpMsg -Points to a structure that contains
                           details about the message.

RETURN    The callback function should return zero.
|
| */
DWORD CALLBACK HookSndMsgProc(int nCode, WORD wParam, LPCALLWNDPROC lpMsg)
{
    if (nCode == HC_ACTION)
    {
        HookMessage(lpMsg->hwnd, lpMsg->wMsg, lpMsg->wParam, lpMsg->lParam);
    }
    return CallNextHookEx(hSndMsgHook, nCode, wParam, (LONG)lpMsg);
}

```

```

/*-----
FUNCTION  void WINAPI HookInstall(fInstall)

DESCRIPTION Set up all necessary hooking code to view all messages.

PARAMETERS BOOL fInstall - Specifies install/uninstall toggle.

```

```

| RETURN    None.
|
| */
void WINAPI HookInstall(BOOL fInstall)
{
    if (fInstall)
    { // Install only if there isn't already a hook installed
        /*
        ** Install hook for posted messages.
        */
        if (!hGetMsgHook)
            hGetMsgHook = SetWindowsHookEx(WH_GETMESSAGE,
(FARPROC)HookGetMsgProc, hInst, NULL);

        /*
        ** Install hook for sent messages.
        */
        if (!hSndMsgHook)
            hSndMsgHook = SetWindowsHookEx(WH_CALLWNDPROC,
(FARPROC)HookSndMsgProc, hInst, NULL);
    }
    else
    {
        UnhookWindowsHookEx(hGetMsgHook);
        UnhookWindowsHookEx(hSndMsgHook);
        hGetMsgHook = NULL;
        hSndMsgHook = NULL;
    }
}

|-----
| FUNCTION    BOOL WINAPI HookJournalBusy(void)
|
| DESCRIPTION Return whether or not the DLL has a journal hook already
|             installed
|
| PARAMETERS None.
|
| RETURN     TRUE if journal busy.
|
| */
BOOL WINAPI HookJournalBusy(void)
{
    return bJournalBusy; // Is journal playback active?
}

|-----
| FUNCTION    static void PlayNotify(void)
|
| DESCRIPTION Notify about end of playyback.
|
| PARAMETERS None.
|

```

| RETURN None.

\*/

static void PlayNotify(void)

{

if (hWndNotify)

{

SendMessage(hWndNotify, wParamNotify, 0, 0L);

}

}

/\*

FUNCTION DWORD CALLBACK PlayProc(nCode, wParam, lParam)

DESCRIPTION The PlayProc function is a callback function that a library can use to insert mouse and keyboard messages into the system message queue.

PARAMETERS int nCode - Specifies whether the callback function should process the message or call the CallNextHookEx function. If this parameter is less than zero, the callback function should pass the message to CallNextHookEx without further processing.

WORD wParam - Specifies a NULL value.

LPEVENTMSG lParam - Points to an EVENTMSG structure that represents the message being processed by the callback function.

RETURN The callback function should return a value that represents the amount of time, in clock ticks, that the system should wait before processing the message. This value can be computed by calculating the difference between the time members of the current and previous input messages. If the function returns zero, the message is processed immediately.

\*/

DWORD CALLBACK PlayProc(int nCode, WORD wParam, LPEVENTMSG lParam)

{

DWORD dwRetcode = NULL;

BOOL bCallNext = TRUE;

LPRECORD lpList;

switch (nCode)

{

case HC\_SKIP :

// See if we are all done playing back

if (!lpJmlList)

{

//OutputDebugString("HC\_SKIP - Next event is NULL so we're all done.\n");

UnhookWindowsHookEx(hJournalHook);

PlayNotify();

bJournalBusy = FALSE;

```

// if (!wNumEvents)
//     OutputDebugString(" Played the number of events recorded.\n");
//     }
//     else {
//         wNumEvents--;

//         lpList = lpJmList->pNext;
//         Gfree(lpJmList);
//         lpJmList = lpList;
//     }
//     bCallNext = FALSE;
//     break;

case HC_GETNEXT :
    // Lock and playback this member of the list.

    if (lpJmList)
    {

        lpMsg->message = lpJmList->msg.message;
        lpMsg->paramL = lpJmList->msg.paramL;
        lpMsg->paramH = lpJmList->msg.paramH;

        switch (sPlaybackSpeed)
        {
            case -1 : // Full Speed
                lpMsg->time = GetTickCount();
                dwRetcode = dwInitPlaybackTime -
GetTickCount() + GetDoubleClickTime() + 1;
                if ((long)dwRetcode < 0) // if time has gone by
return
                    dwRetcode = 0;
                    // 0 for the wait time.
                    break;

            default :
            case 0 : // Original Speed
                lpMsg->time = lpJmList->msg.time +
dwInitPlaybackTime;
                dwRetcode = lpMsg->time - GetTickCount();
                if ((signed long)dwRetcode < 0) // if time has
gone by return
                    dwRetcode = 0;
                    // 0 for the wait time.
                    break;
        }

    }

    bCallNext = FALSE;
    break;

case HC_SYSMODALON :
    // A system modal dialog box has appeared.
    // Something bad must have happened.

```



```
// Free all remaining event structures and unhook.

// Should some sort of error message be displayed to the user when
// we receive the HC_SYSMODALOFF to say that we stopped playback?
```

```
while (lpJmList)
{
    lpList = lpJmList->pNext;
    Gfree(lpJmList);
    lpJmList = lpList;
}

UnhookWindowsHookEx(hJournalHook);
PlayNotify();
bJournalBusy = FALSE;
break;
```

```
default :
    break;
}
```

```
if (bCallNext)
{
    dwRetcode = CallNextHookEx(hJournalHook, nCode, wParam, (LONG)lpMsg);
}

return dwRetcode;
}
```

```
/*-----
FUNCTION void WINAPI Playback(HWND hWnd, WPARAM wParam, LPARAM lParam)
DESCRIPTION Journal Playback Function
PARAMETERS HWND hWnd - Specifies handle to the window
to send notification to.
UINT wParam - Specifies notification message.
short lParam - Specifies speed of playback.
LPCRECT lpRect - Specifies pointer to the events list.
RETURN None.
*/
void WINAPI Playback(HWND hWnd, WPARAM wParam, LPARAM lParam)
{
    if (bJournalBusy)
        return;

    if (lpList == NULL)
        return;

    hWndNotify = hWnd;
    wParamNotify = wParam;
    bJournalBusy = TRUE;
```

09852049-050901  
T06050-64025860

lpJmlList = lpList;  
sPlaybackSpeed = sSpeed;

dwInitPlaybackTime = GetTickCount();  
dwPrevMsgTime = dwInitPlaybackTime;

hJournalHook = SetWindowsHookEx(WH\_JOURNALPLAYBACK, (FARPROC)PlayProc,  
hInst, NULL);  
return;

}

```
/*-----  
FUNCTION    void WINAPI HookFreeJournal(void)  
DESCRIPTION Release journal hook.  
PARAMETERS None.  
RETURN     None.
```

```
*/  
void WINAPI HookFreeJournal(void)  
{  
    if (hJournalHook)  
    {  
        UnhookWindowsHookEx(hJournalHook);  
        bJournalBusy = FALSE;  
        hJournalHook = NULL;  
    }  
}
```

```
/*-----  
FUNCTION    static void RecNotify(void)  
DESCRIPTION Notify about end of recording.  
PARAMETERS None.  
RETURN     None.
```

```
*/  
static void RecNotify(void)  
{  
    LPRECORD lpList;  
    DWORD dwFirstTime;  
  
    // reset the time field in all of these  
    if (lpJmlList)  
        dwFirstTime = lpJmlList->msg.time;  
  
    lpList = lpJmlList;
```

```

while (lpList != NULL)
{
    lpList->msg.time -= dwFirstTime;
    lpList = lpList->pNext;
}

SendMessage(hWndNotify, wParam, 0, (LONG)lpJmlList);

```

```

}

```

```

/*-----

```

```

FUNCTION  DWORD CALLBACK RecProc(nCode, wParam, lParam)

```

```

DESCRIPTION The RecProc function is a callback function that records
messages that the system removes from the system message queue.

```

```

PARAMETERS int nCode    - Specifies whether the callback function
                        should process the message or call the
                        CallNextHookEx function. If this parameter
                        is less than zero, the callback function
                        should pass the message to CallNextHookEx
                        without further processing.

```

```

WORD wParam    - Specifies a NULL value.

```

```

LONG lParam    - Points to an EVENTMSG structure that
represents the message being processed
by the callback function.

```

```

RETURN      The callback function should return zero.

```

```

*/

```

```

DWORD CALLBACK RecProc(int nCode, WORD wParam, LONG lParam)

```

```

{
    static LPRECORD lpPrevList; // Handle to prev recorded event
    static WORD     wNumEvents;  // ** number of events recorded ** for testing **
    static BOOL     bPause = FALSE;
    LPRECORD        lpList;
    LPEVENTMSG       lpEvent;
    BOOL             bCallNext = TRUE;
    DWORD            dwRetcode = 0;
    DWORD            dwTime;

```

```

    switch (nCode)
    {

```

```

        case HC_ACTION :

```

```

            if (bPause)

```

```

            {

```

```

                break;

```

```

            }

```

```

            dwTime = GetTickCount();

```

```

            lpEvent = (LPEVENTMSG) lParam;

```

```

            if (lpEvent->message == WM_KEYDOWN && LOBYTE(lpEvent-

```

```

>paramL) == wStopKey)

```

```

            {

```

```

        HookFreeJournal();
        RecNotify();
        break;
    }
    if (lpEvent->message >= WM_MOUSEFIRST && lpEvent->message <=
WM_MOUSELAST)
    {
        if (wMouRec == REC_MOUIGNORE)
        {
            break;
        }
        else if (wMouRec == REC_MOUCLICK && lpEvent-
>message == WM_MOUSEMOVE)
        {
            break;
        }
    }
    // Allocate the next member (zeroinit it so hNext field doesn't
    // have to be explicitly set to zero)
    lpList = Gmalloc((DWORD) sizeof(RECORD));

    if (lpList == NULL)
    {
        HookFreeJournal();
        RecNotify();
        break;
    }

    // Update the previous member to point to this new one.
    if (lpJmlList == NULL)
    { // It's the first one
        wNumEvents = 0;
        lpJmlList = lpList;
    }
    else
    {
        lpPrevList->pNext = lpList;
    }
    lpPrevList = lpList;

    // Store the message in the new one

    lpList->msg = *lpEvent;
    lpList->msg.time = dwTime;
    break;

case HC_SYSMODALON:
    bPause = TRUE;
    break;

case HC_SYSMODALOFF:
    bCallNext = FALSE;
    bPause = FALSE;
    HookFreeJournal();
    RecNotify();
    break;

```

```

        default :
            break;
    }

    if (bCallNext) {
        dwRetcode = CallNextHookEx(hJournalHook, nCode, wParam, lParam);
    }
    return dwRetcode;
}

```

```

/*-----
FUNCTION void WINAPI Record(hWnd, wParam, wKey, wMou)
DESCRIPTION Journal Record Function
PARAMETERS HWND hWnd - Specifies handle to the window
to send notification to.
        UINT wParam - Specifies notification message.
        UINT wKey - Specifies stop key VK_ value.
        UINT wMou - Specifies type of mouse events that
        should be recorded.
RETURN None.
*/
void WINAPI Record(HWND hWnd, UINT wParam, UINT wKey, UINT wMou)
{

```

```

    if (bJournalBusy)
        return;

```

```

    hWndNotify = hWnd;
    wParamNotify = wParam;
    wStopKey = wKey;
    wMouRec = wMou;
    lpJrnList = NULL;

```

```

    hJournalHook = SetWindowsHookEx(WH_JOURNALRECORD, (FARPROC)RecProc,
hInst, NULL);

```

```

    if (hJournalHook)
        bJournalBusy = TRUE;

```

```

}

```

```

    */
    else
    {
        switch (iScrCom & SCRLS_ACT)
        {
            case SB_LINEUP:
                /* line up
                */
                idWord = IDW_LINEUP;
                break;
            case SB_LINEDOWN:
                /* line down
                */
                idWord = IDW_LINEDOWN;
                break;
            case SB_PAGEUP:
                /* page up
                */
                idWord = IDW_PAGEUP;
                break;
            case SB_PAGEDOWN:
                /* page down
                */
                idWord = IDW_PAGEDOWN;
                break;
        }
    }
    /* MDI frame is a special case
    */
    if (iScrCom & SCRLS_MDI )
    {
        len = wsprintf(
            szCaptionBuf, "%s %s",
            (LPSTR)UserGetDefWord(IDW_MDIFRAME),
            (LPSTR)UserGetDefWord(idWord));
    }
    else
    {
        lstrcpy(szCaptionBuf, UserGetDefWord(idWord));
        len = lstrlen(szCaptionBuf);
    }
    return(len);
}

#ifdef DEBUG_DLG
/*-----
FUNCTION  _LOCAL int ContextPakWindDebug(hwnd)
DESCRIPTION Get debug information for the given window.
PARAMETERS HWND hwnd - Specifies handle to the window we are looking at.
RETURN    Length of the caption text.

```

```

|
| */
| _LOCAL int ContextPakWindDebug(HWND hwnd)
| {
|     /* Now we can recieve text from EDIT
|     */
|     return((int) SendMessage(hwnd, WM_GETTEXT, sizeof(szCaptionBuf) - 1 ,
| (LONG)(LPSTR)szCaptionBuf));
| }

```

```

| /*-----

```

```

| FUNCTION _LOCAL int ContextPakDebug(void)

```

```

| DESCRIPTION Create debug string.

```

```

| PARAMETERS None.

```

```

| RETURN Length of the caption text.

```

```

| */

```

```

| _LOCAL int ContextPakDebug(void)
| {

```

```

|     /* ADD DEBUG INFO TO THE CONTEXT STRING
|     */

```

```

|     HWND hwnd = pciLast->hwnd;
|     PSTR Str;

```

```

|     int len = lstrlen(szCaptionBuf);
|     int lend;

```

```

|     if (! len)
|     {

```

```

|         switch (pciLast->conType)
|         {

```

```

|             case CON_WIND:

```

```

|             case CON_ICON:

```

```

|                 /* Add window debug info
|                 */

```

```

|                 len = ContextPakWindDebug(hwnd);
|                 break;

```

```

|             default:

```

```

|                 ;

```

```

|         }

```

```

|     if (! len)
|     {

```

```

|         /* No text for this item
|         */

```

```

|         lstrcpy(szCaptionBuf, "<No Caption>");
|         len = lstrlen(szCaptionBuf);

```

```

|     }

```

```

| }

```

T06090" 64025860

```

/* Move start pointer
*/
Str = szCaptionBuf + len;

/* Show the handle and the parent handle for the related window.
*/
lend = wsprintf(Str, "\\t%1d %04X\\t", pciLast->iLevel, hwnd);
Str += lend;

/* Add debug info to the string.
*/
switch (pciLast->conType)
{
    case CON_WIND:
    case CON_ICON:
        /* Its a window or a control.
        */
        if (! hwnd)
            /* No associated window ?
            */
            break;

        /* Parent and owner
        */
        lend = wsprintf(Str, "%04X %04X ", GetParent(hwnd), GetWindow(hwnd,
GW_OWNER));

        /* Add the class name to it.
        */
        GetClassName(hwnd, Str+lend, MAXSTRING);

        /* Usefull properties
        */
        if (! IsWindowEnabled(hwnd))
            lstrcat(Str, " <INACTIVE>");
        else if (! IsWindowVisible(hwnd))
            lstrcat(Str, " <INVISIBLE>");
        else if (IsZoomed(hwnd))
            lstrcat(Str, " <MAXIMIZED>");
        else if (IsIconic(hwnd))
            lstrcat(Str, " <MINIMIZED>");
        if (hwnd == GetActiveWindow())
            lstrcat(Str, " <ACTIVE>");
        if (hwnd == GetFocus())
            lstrcat(Str, " ,<FOCUS>");

        /* We need to return this
        */
        lend = lstrlen(Str);
        break;

    case CON_SYSCOM:
        /* System commans
        */
        lend = wsprintf(Str, "<SYSTEM COMMAND %d>", pciLast->u.SysCom);

```



```

        break;

    case CON_MENUPOPUP:
        /* Popup menu properties
        */
        lend = wsprintf(Str, "%04x <POPUP MENU %d>",
            GetMenuState(pciLast->u.MenuPop.hMenu, pciLast-
>u.MenuPop.iEntry,
                MF_BYPOSITION), pciLast->u.MenuPop.iEntry);
        break;

    case CON_MENU:
        /* Menu item properties
        */
        lend = wsprintf(Str, "<MENU ITEM %d>", pciLast->u.Menu.id);
        break;

    case CON_ACCEL:
        /* Accelerator
        */
        lend = wsprintf(Str, "<ACCELERATOR FOR %d>", pciLast->u.Acc.id);
        break;

    case CON_LAUNCH:
        /* ProgMan launch command
        */
        lend = wsprintf(Str, "<%s>", (LPSTR)(pciLast->u.PMItem.szFile));
        break;

    case CON_MACRO:
        /* Macro
        */
        lend = wsprintf(Str, "<MACRO>");
        break;
    }

    /* Calculate maximum length
    */
    if (lend > iDebugCapLen)
        iDebugCapLen = lend;

    return(len);
}

#endif

/*-----
FUNCTION  _LOCAL int ContextPak(void)
DESCRIPTION Build a context string for the context block.
            User pciLast to identify the object.
PARAMETERS None.
RETURN    Length of the caption text.

```

```

/*
** File: PLAYBACK.C
**
** Functions for Macro Execution
**
** Public functions:  MakeHookReady
**                   VCM_Execute
**
** Private Functions : me_SingleCommand
**                   me_Clk
**                   me_Key
**                   me_String
**                   me_Execute
**
*****
*****/

```

```

#define WIN31          // need this to use extended 3.1 functionality
#include <windows.h>

```

```

#include <shellapi.h>
#include <ctype.h>

```

```

#include "vtools.h"
#include "vc.h"

```

```

/* Private Function Prototypes
*/

```

```

_LOCAL BOOL me_SingleCommand(LPMACRO, HWND);
_LOCAL BOOL me_Clk(LPMACRO);
_LOCAL BOOL me_Key(VCM_KEY KeyType);
_LOCAL BOOL me_String(LPSTR Str);
_LOCAL BOOL me_Execute(LPSTR Str);

```

```

/*-----
|
| FUNCTION   BOOL MakeHookReady(void)
|
| DESCRIPTION Wait until we finish playback.
|
| PARAMETERS None.
|
| RETURN    TRUE if success.
|
*/

```

```

BOOL MakeHookReady(void)
{
    MSG msg;

    while (HookJournalBusy())
    {
        if (PeekMessage(&msg, NULL, NULL, NULL, PM_REMOVE))
            ProcessMessage(msg);
    }
}

```

```

        return TRUE;
    }

/*-----
FUNCTION  BOOL VCM_Execute(LPMACRO CmdPtr, HWND hGlobalWnd)
DESCRIPTION Processes the command encoded in the input
            command struture.
PARAMETERS LPMACRO CmdPtr   - Points to an list of MACRO elements.
            HWND   hGlobalWnd - Default window to send commands to.
RETURN     TRUE if success.
*/
BOOL VCM_Execute(LPMACRO CmdPtr, HWND hGlobalWnd)
{
    WORD wErr;
    HWND hLocalWnd;

    /* Check for NULL pointers
    */
    if (CmdPtr == NULL)
        return 1;

    while (CmdPtr != NULL)
    {
        /* use currently active win
        */
        if ((CmdPtr->cmdType == CMD_KEY) ||
            (CmdPtr->cmdType == CMD_TEXT) ||
            (CmdPtr->cmdType == CMD_LAUNCH))
            hLocalWnd = NULL;
        else
            hLocalWnd = hGlobalWnd;

        /* Process a single command
        */
        if (wErr = me_SingleCommand(CmdPtr, hLocalWnd))
            return wErr;

        /* Get the next command
        */
        CmdPtr = CmdPtr->pNext;
    }
    return TRUE;
}

/*-----
FUNCTION  _LOCAL BOOL me_SingleCommand(LPMACRO CmdPtr, HWND hWnd)
DESCRIPTION Execute single macro command.

```

```

PARAMETERS LPMACRO CmdPtr - Points to an list of MACRO elements.
           HWND hGlobalWnd - Default window to send commands to.

RETURN TRUE if success.

*/
LOCAL BOOL me_SingleCommand(LPMACRO CmdPtr, HWND hWnd)
{
    RECT rect;
    POINT pt;

    BOOL bFoundIt;
    HMENU hMenu;
    WORD wTotal, wFlags, i, KeyPos;
    MACRO macro;
    WORD wKeyUp, wKeyDown;
    int iLevel;

    /* Was a specific window given or are we to assume that we should use
    ** the currently active window?
    */
    if (! hWnd)
        hWnd = GetActiveWindow();

    /* Make sure it is a valid window handle
    */
    if (! IsWindow(hWnd))
        return FALSE;

    /* Was a class specified and if so was there also window text given.
    ** Don't allow specification of window text without the window
    ** class being given as well.
    */

    /* Determine the type of command and process the command specific action.
    */
    switch (CmdPtr->cmdType)
    {
        case CMD_MENU :
            /* Verify that the given window has a menu (do I need to bother w/this?)
            */
            if (!(hMenu = GetMenu(hWnd)))
                return FALSE;
            /* Check to make sure selection is available
            */
            i = GetMenuState(hMenu, CmdPtr->Cmd.Menu.id, MF_BYCOMMAND);
            if ((i & MF_DISABLED) || (i & MF_GRAYED) || (i == -1))
                break;

            /* Clear the menus before the command is sent
            */
            iLevel = HookGet_MenuLevel();
            while (iLevel-- >= 0)
            {
                PostMessage(hWnd, WM_SYSKEYDOWN, VK_ESCAPE, 0L);
                Yield();
            }
        }
    }
}

```

}

PostMessage(hWnd, WM\_COMMAND, CmdPtr->Cmd.Menu.id, 0L);  
break;

case CMD\_MENUPOPUP :

/\* Verify that the given window has a menu (do I need to bother w/this?)  
\*/

if (!(hMenu = GetMenu(hWnd)))  
return FALSE;

/\* Test to see where the current menu highlighting is.  
\*/

hMenu = HookGet\_MenuAtLevel(0);

/\* No menu up - Activate the Menu Bar  
\*/

if (!hMenu)  
{

hMenu = GetMenu(hWnd);  
PostMessage(hWnd, WM\_SYSCOMMAND, SC\_KEYMENU,

0L);

i = CmdPtr->Cmd.MenuPopup.iKeyPos;  
while (i--)

0L);

{  
PostMessage(hWnd, WM\_SYSKEYDOWN, VK\_RIGHT,  
Yield();  
}

/\* Need to check to see if there really is a menu to pop up or  
\*\* if it is a menu item on the menu bar that has no pulldown.  
\*/

if ((i = GetMenuItemID(hMenu, CmdPtr->Cmd.MenuPopup.iKeyPos)) != -1)

VK\_ESCAPE, 0L);

{  
iLevel = HookGet\_MenuLevel();  
while (iLevel-- >= 0)  
{  
PostMessage(hWnd, WM\_SYSKEYDOWN,  
Yield();  
}

VK\_DOWN, 0L);

PostMessage(hWnd, WM\_COMMAND, i, 0L);  
}  
else  
PostMessage(hWnd, WM\_SYSKEYDOWN,

}  
/\* It's a cascading popup  
\*/

else  
{

/\* Pop "back" the menus to the correct level  
\*/

VK\_ESCAPE, 0L);

>Cmd.MenuPopup.wLevel);

MF\_SEPARATOR)))

wKeyDown, 0L);

```
while(HookGet_Menu(CmdPtr->Cmd.MenuPopup.wLevel + 1))
{
    PostMessage(hWnd, WM_SYSKEYDOWN,
        Yield();
}

/* Get the current position that is hilited
*/
hMenu = HookGet_MenuAtLevel(CmdPtr-
wTotal = GetMenuItemCount(hMenu);

i = 0;
KeyPos = 0;
bFoundIt = FALSE;
while ((i < wTotal) && !bFoundIt)
{
    wFlags = GetMenuState(hMenu, i, MF_BYPOSITION);
    if (wFlags & MF_HILITE)
        bFoundIt = TRUE;
    else
    {
        if ((wFlags & MF_POPUP) || (!(wFlags &
            KeyPos++;
            i++;
        }
    }

/* Must take separators into account in position
*/
i = KeyPos;

if (CmdPtr->Cmd.MenuPopup.wLevel)
{
    wKeyUp = VK_UP;
    wKeyDown = VK_DOWN;
}
else
{
    wKeyUp = VK_LEFT;
    wKeyDown = VK_RIGHT;
}

if (i < (WORD)CmdPtr->Cmd.MenuPopup.iKeyPos)
{
    i = CmdPtr->Cmd.MenuPopup.iKeyPos - i;
    while (i--> 0)
    {
        PostMessage(hWnd, WM_SYSKEYDOWN,

    }
}
else
{
```

```

        if (i > (WORD)CmdPtr->Cmd.MenuPopup.iKeyPos)
        {
            i = i - CmdPtr->Cmd.MenuPopup.iKeyPos;
            while (i--)
            {
                PostMessage(hWnd,
WM_SYSKEYDOWN, wKeyUp, 0L);
            }
        }
        PostMessage(hWnd, WM_SYSKEYDOWN, VK_RETURN, 0L);
    }
    break;

    case CMD_SYSTEM :
        if ((CmdPtr->Cmd.System.wCmd == SC_KEYMENU) || (CmdPtr-
>Cmd.System.wCmd == SC_MOUSEMENU))
        {
            /* Activating the system menu of an iconized window can't be
done
            ** with the normal syscommands and syskeys.
effect
            ** Using mouse commands works but it has the unpleasant side
acceptable
            ** of moving the pointer. Therefore this may not be an
            ** solution.
            */
            if (GetParent(hWnd))
            {
                /* This combination seems to work in all cases except
                ** the system menu of a child window in Excel that is
                */
                PostMessage(hWnd, WM_SYSCOMMAND, CmdPtr-
                PostMessage(hWnd, WM_SYSKEYDOWN,
VK_RETURN, 0L);
            }
            else
            {
                PostMessage(hWnd, WM_SYSCOMMAND,
                PostMessage(hWnd, WM_SYSKEYDOWN,
SC_KEYMENU, 0L);
                PostMessage(hWnd, WM_SYSKEYDOWN,
VK_SPACE, 0L);
            }
        }
        else
        {
            iLevel = HookGet_MenuLevel();
            while (iLevel-- >= 0)
            {

```

```

VK_ESCAPE, 0L);
        PostMessage(hWnd, WM_SYSKEYDOWN,
        Yield();
    }

    PostMessage(hWnd, WM_SYSCOMMAND, CmdPtr-
>Cmd.System.wCmd, 0L);
    }

    break;

case CMD_MESSAGE :
    /* Just message to post
    */
    PostMessage(hWnd, CmdPtr->Cmd.Msg.wMsg, CmdPtr-
>Cmd.Msg.wParam, CmdPtr->Cmd.Msg.lParam);
    break;

case CMD_SELECT :
{
    /* Bring hWnd to the top and activate it.
    */
    POINT pt;
    int i;

    if (GetWindowLong(hWnd, GWL_STYLE) & WS_CHILD)
    {
        SetFocus(hWnd);
        GetWindowRect(hWnd, &rect);

        pt.x = rect.left;
        pt.y = rect.top;
        for (i = 0; i < 5; i++)
        {
            if (WindowFromPoint(pt) == hWnd)
                break;
            pt.x++;
            pt.y++;
        }
        macro.cmdType = CMD_MOUSE;
        macro.pNext = NULL;
        macro.szWndClass = NULL;
        macro.szDesc = NULL;
        macro.Cmd.Mouse.mouType = MOU_LBCLK;
        macro.Cmd.Mouse.bPosType = VCM_MP_SCREEN;
        macro.Cmd.Mouse.wX = pt.x;
        macro.Cmd.Mouse.wY = pt.y;
        macro.Cmd.Mouse.CtrlPressed = 0;
        macro.Cmd.Mouse.ShiftPressed = 0;
        macro.Cmd.Mouse.AltPressed = 0;

        VCM_Execute(&macro, hWnd);
    }
    else
    {
        BringWindowToTop(hWnd);
    }
}

```



```

    }
    break;
}

```

```

/* Mouse, Keyboard, and Journal Playback commands will all be handled via
** a Journal Playback Hook. We still need to go through the window
** checking above to make sure that if the events are to go to a specific
** window that the window is there.
*/

```

```

case CMD_MOUSE :

```

```

/* For all mouse commands, convert any client coordinates
** to screen coordinates before proceeding further.
*/
if (CmdPtr->Cmd.Mouse.bPosType == VCM_MP_CLIENT)
{
    pt.x = CmdPtr->Cmd.Mouse.wX;
    pt.y = CmdPtr->Cmd.Mouse.wY;
    ClientToScreen(hWnd, (LPPOINT) &pt);
    CmdPtr->Cmd.Mouse.wX = pt.x;
    CmdPtr->Cmd.Mouse.wY = pt.y;
    /* in case it's used later
    */
    CmdPtr->Cmd.Mouse.bPosType = VCM_MP_SCREEN;
}

```

```

switch (CmdPtr->Cmd.Mouse.mouType)
{

```

```

    case MOU_MOVE :

```

```

        /* Do moves need to be done via playback or is this

```

```

        */

```

```

        SetCursorPos(CmdPtr->Cmd.Mouse.wX, CmdPtr-

```

OK???

>Cmd.Mouse.wY);

```

        break;

```

```

/* Is it necessary to set the focus for clicks and double clicks?
*/

```

```

case MOU_LDBLCLK : // Double Clicks

```

```

case MOU_RDBLCLK :

```

```

case MOU_MDBLCLK :

```

```

case MOU_LBCLK : // Single Clicks

```

```

case MOU_RBCLK :

```

```

case MOU_MBCLK :

```

```

    return (me_Clk(CmdPtr));

```

```

    break;

```

```

}

```

```

break;

```

```

case CMD_KEY :

```

```

/* May need more values passed in for the OEM scan code to be set.

```

```

** Is it necessary to set the focus here before the key is sent?

```

09852049-050901

```

    /** if window is iconized or ALT is pressed then WM_SYSKEY
    */

```

```

    return (me_Key(CmdPtr->Cmd.Key));

```

```

    break;

```

```

case CMD_TEXT :

```

```

    return (me_String(CmdPtr->szDesc));

```

```

    break;

```

```

case CMD_LAUNCH :

```

```

    return (me_Execute(CmdPtr->szDesc));

```

```

    break;

```

```

case CMD_JOURNAL :

```

```

{

```

```

    LPRECORD pFirstRecord;

```

```

    LPRECORD pRecord;

```

```

    POINT pt;

```

```

    if (HookJournalBusy())

```

```

        return FALSE;

```

```

    /* need to define how the playback list is going to be sent and what
    ** we are going to do about any timing type problems such as windows
    ** taking longer to appear than they did in the original recording etc.
    */

```

```

    pFirstRecord = RecordMake(CmdPtr->Cmd.Journal.pRecord);

```

```

    for (pRecord = pFirstRecord; pRecord != NULL; pRecord = pRecord-

```

```

    >pNext)

```

```

    {

```

```

        if (pRecord->msg.message >= WM_MOUSEFIRST &&
        pRecord->msg.message <= WM_MOUSELAST)

```

```

        {

```

```

            pt.x = pRecord->msg.paramL;

```

```

            pt.y = pRecord->msg.paramH;

```

```

            ClientToScreen(hWnd, &pt);

```

```

            pRecord->msg.paramL = pt.x;

```

```

            pRecord->msg.paramH = pt.y;

```

```

        }

```

```

    }

```

```

    Playback(NULL, 0, 0, pFirstRecord);

```

```

    break;

```

```

}

```

```

default :

```

```

    /* error - Unknown Command Type

```

```

    */

```

```

    return FALSE;

```

```

    break;

```

```

}

```

T06050" 64025860

```

    return TRUE;
}

```

```

#define MAKEKEY(uVKey) (MAKELONG(uVKey, MapVirtualKey(uVKey, 0)))

```

```

/*-----
FUNCTION  _LOCAL BOOL me_Clk(LPMACRO CmdPtr)

DESCRIPTION Execute mouse macro command.

PARAMETERS LPMACRO CmdPtr  - Points to an list of MACRO elements.

RETURN    TRUE if success.
*/
_LOCAL BOOL me_Clk(LPMACRO CmdPtr)
{
    LPRECORD lpList, lpHead;
    WORD  Down, DownSec, Up;
    WORD  time = 0x50;
    BOOL  bSysKey = (CmdPtr->Cmd.Mouse.AltPressed) && ! (CmdPtr-
>Cmd.Mouse.CtrlPressed);
    POINT ptCur;

    GetCursorPos(&ptCur);

    /* Mouse coordinates have already been converted to screen coordinates
    */
    switch (CmdPtr->Cmd.Mouse.mouType)
    {
        case MOU_LBCLK :
            Down = WM_LBUTTONDOWN;
            DownSec = NULL;
            Up = WM_LBUTTONUP;
            break;
        case MOU_RBCLK :
            Down = WM_RBUTTONDOWN;
            DownSec = NULL;
            Up = WM_RBUTTONUP;
            break;
        case MOU_MBCLK :
            Down = WM_MBUTTONDOWN;
            DownSec = NULL;
            Up = WM_MBUTTONUP;
            break;
        case MOU_LBDBCLK :
            Down = WM_LBUTTONDOWN;
            DownSec = WM_LBUTTONDBLCLK;
            Up = WM_LBUTTONUP;
            break;
        case MOU_RBDBCLK :
            Down = WM_RBUTTONDOWN;
            DownSec = WM_RBUTTONDBLCLK;
    }
}

```

T05090" 64025860

```

        Up = WM_RBUTTONDOWN;
        break;
    case MOU_MDBLCLK :
        Down = WM_MBUTTONDOWN;
        DownSec = WM_MBUTTONDOWNBLCLK;
        Up = WM_MBUTTONUP;
        break;
    default:
        return FALSE;
}

lpList = Gmalloc((DWORD) sizeof(RECORD));
lpHead = lpList;

if (lpList)
{
    lpList->msg.message = WM_MOUSEMOVE;
    lpList->msg.paramL = CmdPtr->Cmd.Mouse.wX;
    lpList->msg.paramH = CmdPtr->Cmd.Mouse.wY;
    lpList->msg.time = time;
    time += 0x50;
}
else
    return FALSE;

if (CmdPtr->Cmd.Mouse.AltPressed)
{
    lpList->pNext = Gmalloc((DWORD) sizeof(RECORD));
    if (lpList->pNext)
    {
        lpList = lpList->pNext;
        lpList->msg.message = WM_SYSKEYDOWN;
        lpList->msg.paramL = MAKEKEY(VK_MENU);
        lpList->msg.paramH = 0x1; // repeat count
        lpList->msg.time = time;
        time += 0x50;
    }
    else
        return FALSE;
}

if (CmdPtr->Cmd.Mouse.CtrlPressed)
{
    lpList->pNext = Gmalloc((DWORD) sizeof(RECORD));

    if (lpList->pNext)
    {
        lpList = lpList->pNext;
        lpList->msg.message = WM_KEYDOWN;
        lpList->msg.paramL = MAKEKEY(VK_CONTROL);
        lpList->msg.paramH = 0x1; // repeat count
        lpList->msg.time = time;
        time += 0x50;
    }
    else

```

```

        return FALSE;
    }

    if (CmdPtr->Cmd.Mouse.ShiftPressed)
    {
        lpList->pNext = Gmalloc((DWORD) sizeof(RECORD));

        if (lpList->pNext)
        {
            lpList = lpList->pNext;
            lpList->msg.message = bSysKey ? WM_SYSKEYDOWN :
WM_KEYDOWN;

            lpList->msg.paramL = MAKEKEY(VK_SHIFT);
            lpList->msg.paramH = 0x1; // repeat count
            lpList->msg.time = time;
            time += 0x50;
        }
        else
            return FALSE;
    }

    lpList->pNext = Gmalloc((DWORD) sizeof(RECORD));

    if (lpList->pNext)
    {
        lpList = lpList->pNext;
        lpList->msg.message = Down;
        lpList->msg.paramL = CmdPtr->Cmd.Mouse.wX;
        lpList->msg.paramH = CmdPtr->Cmd.Mouse.wY;
        lpList->msg.time = time;
        time += 0x50;
    }
    else
        return FALSE;

    if (DownSec)
    {
        lpList->pNext = Gmalloc((DWORD) sizeof(RECORD));

        if (lpList->pNext)
        {
            lpList = lpList->pNext;
            lpList->msg.message = Up;
            lpList->msg.paramL = CmdPtr->Cmd.Mouse.wX;
            lpList->msg.paramH = CmdPtr->Cmd.Mouse.wY;
            lpList->msg.time = time;
            time += 0x50;
        }
        else
            return FALSE;

        lpList->pNext = Gmalloc((DWORD) sizeof(RECORD));

        if (lpList->pNext)
        {

```

```

        lpList = lpList->pNext;
        lpList->msg.message = DownSec;
        lpList->msg.paramL = CmdPtr->Cmd.Mouse.wX;
        lpList->msg.paramH = CmdPtr->Cmd.Mouse.wY;
        lpList->msg.time = time;
        time += 0x50;
    }
    else
        return FALSE;
}
lpList->pNext = Gmalloc((DWORD) sizeof(RECORD));

if (lpList->pNext)
{
    lpList = lpList->pNext;
    lpList->msg.message = Up;
    lpList->msg.paramL = CmdPtr->Cmd.Mouse.wX;
    lpList->msg.paramH = CmdPtr->Cmd.Mouse.wY;
    lpList->msg.time = time;
    time += 0x50;
}
else
    return FALSE;

if (CmdPtr->Cmd.Mouse.ShiftPressed)
{
    lpList->pNext = Gmalloc((DWORD) sizeof(RECORD));

    if (lpList->pNext)
    {
        lpList = lpList->pNext;
        lpList->msg.message = bSysKey ? WM_SYSKEYUP : WM_KEYUP;
        lpList->msg.paramL = MAKEKEY(VK_SHIFT);
        lpList->msg.paramH = 0x1; // repeat count
        lpList->msg.time = time;
        time += 0x50;
    }
    else
        return FALSE;
}

if (CmdPtr->Cmd.Mouse.CtrlPressed)
{
    lpList->pNext = Gmalloc((DWORD) sizeof(RECORD));

    if (lpList->pNext)
    {
        lpList = lpList->pNext;
        lpList->msg.message = WM_KEYUP;
        lpList->msg.paramL = MAKEKEY(VK_CONTROL);
        lpList->msg.paramH = 0x1; // repeat count
        lpList->msg.time = time;
        time += 0x50;
    }
    else

```

```

        return FALSE;
    }

    if (CmdPtr->Cmd.Mouse.AltPressed)
    {
        lpList->pNext = Gmalloc((DWORD) sizeof(RECORD));

        if (lpList->pNext)
        {
            lpList = lpList->pNext;
            lpList->msg.message = WM_KEYUP;
            lpList->msg.paramL = MAKEKEY(VK_MENU);
            lpList->msg.paramH = 0x1; // repeat count
            lpList->msg.time = time;
            time += 0x50;
        }
        else
            return FALSE;
    }

    lpList->pNext = Gmalloc((DWORD) sizeof(RECORD));

    if (lpList->pNext)
    {
        lpList = lpList->pNext;
        lpList->msg.message = WM_MOUSEMOVE;
        lpList->msg.paramL = ptCur.x;
        lpList->msg.paramH = ptCur.y;
        lpList->msg.time = time;
        time += 0x50;
    }
    else
        return FALSE;

    if (! MakeHookReady())
        return FALSE;
    else
        Playback(NULL, 0, -1, lpHead);

    return TRUE;
}

/*-----
FUNCTION  _LOCAL BOOL me_Key(KeyType)
DESCRIPTION Execute key macro command.
PARAMETERS VCM_KEY KeyType - Specifies ke description struct.
RETURN    TRUE if success.
*/
_LOCAL BOOL me_Key(VCM_KEY KeyType)
{

```

```

LPRECORD lpList, lpHead;
WORD    time = 0x50;
BOOL    bSysKey = (KeyType.AltPressed) && ! (KeyType.CtrlPressed);
POINT   ptCur;

```

```

GetCursorPos(&ptCur);

```

```

/* Not quite sure why something like a mouse move must be sent
** before the key down to have the key down be recognized.
*/

```

```

lpList = Gmalloc((DWORD) sizeof(RECORD));

```

```

lpHead = lpList;

```

```

if (lpList)

```

```

{
    lpList->msg.message = WM_MOUSEMOVE;
    lpList->msg.paramL = ptCur.x;
    lpList->msg.paramH = ptCur.y;
    lpList->msg.time = time;
    time += 0x50;
}

```

```

else

```

```

    return FALSE;

```

```

if (KeyType.AltPressed)

```

```

{
    lpList->pNext = Gmalloc((DWORD) sizeof(RECORD));

```

```

    if (lpList->pNext)

```

```

    {
        lpList = lpList->pNext;
        lpList->msg.message = WM_SYSKEYDOWN;
        lpList->msg.paramL = MAKEKEY(VK_MENU);
        lpList->msg.paramH = 0x1; // repeat count
        lpList->msg.time = time;
        time += 0x50;
    }

```

```

    else

```

```

        return FALSE;

```

```

}

```

```

if (KeyType.CtrlPressed)

```

```

{
    lpList->pNext = Gmalloc((DWORD) sizeof(RECORD));

```

```

    if (lpList->pNext)

```

```

    {
        lpList = lpList->pNext;
        lpList->msg.message = WM_KEYDOWN;
        lpList->msg.paramL = MAKEKEY(VK_CONTROL);
        lpList->msg.paramH = 0x1; // repeat count
        lpList->msg.time = time;
        time += 0x50;
    }

```

```

    else

```



```

        return FALSE;
    }

    if (KeyType.ShiftPressed)
    {
        lpList->pNext = Gmalloc((DWORD) sizeof(RECORD));

        if (lpList->pNext)
        {
            lpList = lpList->pNext;
            lpList->msg.message = bSysKey ? WM_SYSKEYDOWN :
WM_KEYDOWN;
            lpList->msg.paramL = MAKEKEY(VK_SHIFT);
            lpList->msg.paramH = 0x1; // repeat count
            lpList->msg.time = time;
            time += 0x50;
        }
        else
            return FALSE;
    }

    if (KeyType.cKey)
    {
        lpList->pNext = Gmalloc((DWORD) sizeof(RECORD));

        if (lpList->pNext)
        {
            lpList = lpList->pNext;
            lpList->msg.message = bSysKey ? WM_SYSKEYDOWN :
WM_KEYDOWN;
            lpList->msg.paramL = MAKEKEY(KeyType.cKey);
            lpList->msg.paramH = 0x1; // repeat count
            lpList->msg.time = time;
            time += 0x50;
        }
        else
            return FALSE;

        lpList->pNext = Gmalloc((DWORD) sizeof(RECORD));

        if (lpList->pNext)
        {
            lpList = lpList->pNext;
            lpList->msg.message = bSysKey ? WM_SYSKEYUP : WM_KEYUP;
            lpList->msg.paramL = MAKEKEY(KeyType.cKey);
            lpList->msg.paramH = 0x1; // repeat count
            lpList->msg.time = time;
            time += 0x50;
        }
        else
            return FALSE;
    }

    if (KeyType.ShiftPressed)
    {

```

```

lpList->pNext = Gmalloc((DWORD) sizeof(RECORD));

if (lpList->pNext)
{
    lpList = lpList->pNext;
    lpList->msg.message = bSysKey ? WM_SYSKEYUP : WM_KEYUP;
    lpList->msg.paramL = MAKEKEY(VK_SHIFT);
    lpList->msg.paramH = 0x1; // repeat count
    lpList->msg.time = time;
    time += 0x50;
}
else
    return FALSE;
}

if (KeyType.CtrlPressed)
{
    lpList->pNext = Gmalloc((DWORD) sizeof(RECORD));

    if (lpList->pNext)
    {
        lpList = lpList->pNext;
        lpList->msg.message = WM_KEYUP;
        lpList->msg.paramL = MAKEKEY(VK_CONTROL);
        lpList->msg.paramH = 0x1; // repeat count
        lpList->msg.time = time;
        time += 0x50;
    }
    else
        return FALSE;
}

if (KeyType.AltPressed)
{
    lpList->pNext = Gmalloc((DWORD) sizeof(RECORD));

    if (lpList->pNext)
    {
        lpList = lpList->pNext;
        lpList->msg.message = (
            ! (KeyType.cKey) ||
            ! (KeyType.CtrlPressed) ||
            ! (KeyType.ShiftPressed)
        ) ? WM_SYSKEYUP : WM_KEYUP;
        lpList->msg.paramL = MAKEKEY(VK_MENU);
        lpList->msg.paramH = 0x1; // repeat count
        lpList->msg.time = time;
        time += 0x50;
    }
    else
        return FALSE;
}

if (! MakeHookReady())
    return FALSE;
else

```

```

        Playback(NULL, 0, -1, lpHead);

    return TRUE;
}

/*-----
FUNCTION  _LOCAL BOOL me_String(LPSTR Str)
DESCRIPTION Execute string macro command.
PARAMETERS LPSTR Str - Specifies source string.
RETURN    TRUE if success.
*/
_LOCAL BOOL me_String(LPSTR Str)
{
    LPRECORD lpList, lpHead;
    POINT ptCur;
    LONG time=0x50;

    if (Str == NULL)
        return FALSE;

    GetCursorPos(&ptCur);

    /* Not quite sure why something like a mouse move must be sent
    ** before the key down to have the key down be recognized.
    */
    lpList = Gmalloc((DWORD) sizeof(RECORD));
    lpHead = lpList;

    if (lpList)
    {
        lpList->msg.message = WM_MOUSEMOVE;
        lpList->msg.paramL = ptCur.x;
        lpList->msg.paramH = ptCur.y;
        lpList->msg.time = 0x50;
    }
    else
        return FALSE;

    while (*Str != NULL)
    {
        if (isupper(*Str))
        {
            lpList->pNext = Gmalloc((DWORD) sizeof(RECORD));

            if (lpList->pNext)
            {
                lpList = lpList->pNext;
                lpList->msg.message = WM_KEYDOWN;
                lpList->msg.paramL = MAKEKEY(VK_SHIFT);
            }
        }
    }
}

```

```

        lpList->msg.paramH = 0x1; // repeat count
        lpList->msg.time = time+=0x20;
    }
    else
        return FALSE;
}

lpList->pNext = Gmalloc((DWORD) sizeof(RECORD));

if (lpList->pNext)
{
    lpList = lpList->pNext;
    lpList->msg.message = WM_KEYDOWN;
    lpList->msg.paramL = MAKEKEY(toupper(*Str));
    lpList->msg.paramH = 0x1; // repeat count
    lpList->msg.time = time+=0x20;
}
else
    return FALSE;

lpList->pNext = Gmalloc((DWORD) sizeof(RECORD));

if (lpList->pNext)
{
    lpList = lpList->pNext;
    lpList->msg.message = WM_KEYUP;
    lpList->msg.paramL = MAKEKEY(toupper(*Str));
    lpList->msg.paramH = 0x1; // repeat count
    lpList->msg.time = time+=0x20;
}
else
    return FALSE;

if (isupper(*Str))
{
    lpList->pNext = Gmalloc((DWORD) sizeof(RECORD));

    if (lpList->pNext)
    {
        lpList = lpList->pNext;
        lpList->msg.message = WM_KEYUP;
        lpList->msg.paramL = MAKEKEY(VK_SHIFT);
        lpList->msg.paramH = 0x1; // repeat count
        lpList->msg.time = time+=0x20;
    }
    else
        return FALSE;
}

Str++;
}

if (! MakeHookReady())
    return FALSE;
else

```

```
        Playback(NULL, 0, -1, lpHead);

    return TRUE;
}

/*-----
FUNCTION  _LOCAL BOOL me_Execute(LPSTR Str)
DESCRIPTION Execute launch macro command.
PARAMETERS LPSTR Str - Specifies command string
RETURN    TRUE if success.
*/
_LOCAL BOOL me_Execute(LPSTR Str)
{
    char szExec[MAXFILENAME + 1];
    char * pszParam;

    lstrcpy(szExec, Str);
    for (pszParam = szExec; *pszParam != '\0' ; pszParam++)
    {
        if (*pszParam == ' ')
        {
            *pszParam = '\0';
            pszParam++;
            break;
        }
    }
    if (ShellExecute(NULL, NULL, (LPSTR)szExec, (LPSTR)pszParam, NULL,
        SW_SHOWNORMAL) < 32)
    {
        Error(ERRAppExec, (LPSTR)Str);
        return FALSE;
    }

    return TRUE;
}
```

```

/*
** File: STATUS.C
**
** This is the windows display interface for the status window.
**
** Public functions:  StatusSetPref
**                   PhraseListAdd
**                   StatusInit
**                   StatusCheckMsg
**                   StatusGetWindow
**
** Exported functions: PhraseTimerProc
**                   StatusWndProc
**
** Private functions: StatusBarPer
**                   StatusBarDraw
**                   StatusBars
**                   StatusChange
**                   PhraseFind
**                   CloseCallFind
**                   PhraseListMove
**                   PhraseListInc
**                   PhraseListSetup
**                   PhraseDrawitem
**                   PhraseExec
**                   StartTimer
**                   StopTimer
**                   SelectOurFont
**
*****
*****/

#define WIN31          // need this to use extended 3.1 functionality
#include <windows.h>

#include <memory.h>
#include <stdlib.h>

#include "vtools.h"

#include "vc.h"
#include "vcrc.h"          /* only files included by vc.rc */
#include "vchelp.h"        /* only file included by vchlp.hpj */

#define PROMPT_LEN 14
#define IDT_PHRASE 1
#define IDLIST_PHRASE 4
#define BMP_SIZE 16

/*-----
|
| Menu
|
*/
enum
{

```



```

_LOCAL BOOL bCloseCallWas = FALSE;
_LOCAL UINT wCloseCallNumber;
_LOCAL UINT wUttCloseCall = 0;

```

```

_LOCAL int iStatusSizeMin;
_LOCAL BOOL bPause = FALSE;

```

```

_LOCAL HICON hicoMain;
_LOCAL HICON hicoStat;

```

```

_LOCAL HBITMAP hbmpPaint;
_LOCAL HBITMAP hbmpAnd;

```

```

_LOCAL HFONT hFontCur = NULL;
_LOCAL int cxStatusText;
_LOCAL int cyStatusText;

```

```

_LOCAL RECOGRES vrState;

```

```

/*-----
FUNCTION  _LOCAL int StatusBarPer(Rect, val)

DESCRIPTION Return pixel location of a percentage of the rectangle.

PARAMETERS LPRECT Rect - Specifies pointer to the rectangle.
            int val - Specifies value in persents.

RETURN     The pixel location of a percentage of the rectangle.

*/
_LOCAL int StatusBarPer(LPRECT Rect, int val)
{
    return(Rect->left + (int)((((LONG) val) * ((LONG)(Rect->right - Rect->left))) / 100L));
}

```

```

/*-----
FUNCTION  _LOCAL void StatusBarDraw(hDC, Rect, Min, Max, Cur, hBrush)

DESCRIPTION Draw the percentage bar for the current value.

PARAMETERS HDC hDC - Specifies target DC.
            LPRECT Rect - Specifies pointer to the rectangle.
            int Min - Specifies
            int Max - Specifies
            int Cur - Specifies
            HBRUSH hBrush - Specifies

RETURN     None.

*/
_LOCAL void StatusBarDraw(HDC hDC, LPRECT Rect, int Min, int Max, int Cur, HBRUSH
hBrush) {

```



```

HBRUSH hBrBad;
HBRUSH hBrGood;
HANDLE hPrv;
int Maxp;
int Minp;

hBrBad = CreateSolidBrush( RGB(255, 0, 0) );    /* Bad range. */
hBrGood = CreateSolidBrush( RGB(0, 255, 0) );    /* Good range. */

hPrv = SelectObject(hDC, hBrBad) ;

Minp = StatusBarPer(Rect, Min);
if (Min)
{
    Rectangle(hDC, Rect->left, Rect->top, Minp, Rect->bottom);
}

Maxp = StatusBarPer(Rect, Max);
if (Max != 100)
{
    Rectangle(hDC, Maxp, Rect->top, Rect->right, Rect->bottom);
}

SelectObject(hDC, hBrGood) ;
Rectangle(hDC, Minp, Rect->top, Maxp, Rect->bottom);

SelectObject(hDC, hPrv);    /* restore previous selected object. */
DeleteObject(hBrGood) ;
DeleteObject(hBrBad) ;

/*
** Draw the current bar.
*/
hPrv = SelectObject(hDC, hBrush) ;

Minp = Rect->top + ((Rect->bottom - Rect->top) / 4);
Maxp = Rect->top + (((Rect->bottom - Rect->top) * 3) / 4);

Rectangle(hDC, Rect->left, Minp, StatusBarPer(Rect, Cur), Maxp);
SelectObject(hDC, hPrv);    /* restore previous selected object. */
}

```

```

/*-----
FUNCTION  _LOCAL void StatusBars(hDC)

DESCRIPTION Update the data changes to the status window bars.

PARAMETERS HDC hDC - Specifies target DC.

RETURN    None.

*/
_LOCAL void StatusBars(HDC hDC)

```

```

{
    RECT rc;
    HBRUSH hBrush;
    HANDLE hFont;
    COLORREF hOldBk;
    char szWork[PROMPT_LEN + 1];

    if (! (UserGetFlags() & PREF_Confid) && ! (UserGetFlags() & PREF_Volume))
        return;
    if (IsIconic(hwndStatus))
        return;

    /*
    ** Get the new font.
    */
    hFont = SelectObject(hDC, hFontCur);

    /*
    ** Get the location of the status bars.
    ** From the client area rectangle get the rectangle for the first bar.
    */

    GetClientRect(hwndStatus, (LPRECT)&rc);

    DrawIcon(hDC, rc.right - GetSystemMetrics(SM_CXICON) - 2, 2, bPause ? hicoMain :
hicoStat);

    rc.left = PROMPT_LEN * cxStatusText;
    rc.right -= GetSystemMetrics(SM_CXICON) + 4;
    rc.top = 2;
    rc.bottom = cyStatusText;

    /*
    ** Always using this brush.
    */
    hBrush = CreateSolidBrush(RGB(0, 0, 255)); /* Current val */
    hOldBk = SetBkColor(hDC, GetSysColor(COLOR_BTNFACE));

    if (UserGetFlags() & PREF_Confid)
    {
        /*
        ** The confidence display bar.
        */
        LoadString(VChInst, IDS_CONFID, (LPSTR)szWork, PROMPT_LEN);
        TextOut(hDC, cxStatusText, rc.top, szWork, lstrlen(szWork));

        StatusBarDraw(hDC, &rc, UserGetConfidence(), 100,
vrState.confidence, hBrush);

        /*
        ** Move the rectangle down.
        */
        rc.top += cyStatusText + 4;
        rc.bottom += cyStatusText + 4;
    }
}

```

```

if (UserGetFlags() & PREF_Volume)
{
    /*
    ** The volume display bar.
    */
    LoadString(VChInst, IDS_VOLUME, (LPSTR)szWork, PROMPT_LEN);
    TextOut(hDC, cxStatusText, rc.top, szWork, lstrlen(szWork));

    StatusBarDraw(hDC, &rc, iVolumeMin, iVolumeMax,
        vrState.amplitude, hBrush);

    /*
    ** Move the rectangle down.
    */
    rc.top    += cyStatusText + 4;
    rc.bottom += cyStatusText + 4;
}

/*
** Put the old font back.
*/
SelectObject(hDC, hFont);

/*
** Free brush.
*/
DeleteObject(hBrush) ;
SetBkColor(hDC, hOldBk);
}

```

```

/*-----
FUNCTION    _LOCAL void StatusChange(void)

DESCRIPTION Update status information.

PARAMETERS None.

RETURN     None.

*/
_LOCAL void StatusChange(void)
{

    HDC hDC;
    char  szWork[MAXSTRING + 1];

    if (vrState.confidence >= UserGetConfidence())
    {
        StringLoadParam(szWork, IDS_NEW, (LPSTR)vrState.word[0]);
    }
    else
    {

```

```

        LoadString(VChInst, IDS_QUERY, szWork, MAXSTRING);
    }
    SetWindowText(hwndStatus, szWork);

    hDC = GetDC(hwndStatus);

    StatusBars(hDC);

    ReleaseDC(hwndStatus, hDC);
}

/*-----
FUNCTION  _LOCAL UINT PhraseFind(szStr)
DESCRIPTION Find phrase in phrase listbox
PARAMETERS PSTR szStr - Specifies pointer to the phrase.
RETURN    Index in the listbox or LB_ERR.
*/
_LOCAL UINT PhraseFind(PSTR szStr)
{
    UINT wIdx;
    LONG IRet;
    char szWord[MAX_SYMBOL_LENGTH];

    wIdx = 0;
    while (1)
    {
        IRet = SendMessage(hwndList, LB_GETTEXT, wIdx, (LONG)(LPSTR)szWord);
        if (IRet == LB_ERR || IRet == NULL)
            return((UINT)LB_ERR);
        if (! lstrcmpi(szStr, szWord))
            return(wIdx);
        wIdx++;
    }
}

/*-----
FUNCTION  _LOCAL UINT CloseCallFind(szStr)
DESCRIPTION Check phrase as a close call number.
PARAMETERS PSTR szStr - Specifies pointer to the phrase.
RETURN    Index in the listbox or LB_ERR.
*/
_LOCAL UINT CloseCallFind(PSTR szStr)
{
    UINT wIdx;
    LONG IRet;

```

```

UINT wordNum;

for (wIdx = 0; wIdx < wCloseCallNumber; wIdx++)
{
    wordNum = wIdx + '1';
    if (! lstrcmpi(szStr, (char *) &wordNum))
    {
        IRet = SendMessage(hwndList, LB_GETITEMDATA, wIdx, NULL);
        if (IRet == LB_ERR || IRet == NULL)
            continue;
        return(wIdx);
    }
}
return((UINT)LB_ERR);
}

/*-----
FUNCTION  _LOCAL void PhraseListMove(szStr)
DESCRIPTION Move phrase to the close call list.
PARAMETERS PSTR szStr - Specifies pointer to the phrase.
RETURN     None.
*/
_LOCAL void PhraseListMove(PSTR szStr)
{
    int wIdx;
    WORD wordNum;
    char szWord[MAX_SYMBOL_LENGTH];
    LONG IData;

    if (lstrlen (szStr) == 0)
        return;
    wIdx = PhraseFind(szStr);
    if (wIdx == - 1)
        return;
    SendMessage(hwndList, LB_GETTEXT, wIdx, (LONG)(LPSTR)szWord);
    IData = SendMessage(hwndList, LB_GETITEMDATA, wIdx, NULL);
    SendMessage(hwndList, LB_DELETETEXT, wIdx, NULL);
    SendMessage(hwndList, LB_INSERTSTRING, wCloseCallNumber, (DWORD)(LPSTR)
szWord);
    SendMessage(hwndList, LB_SETITEMDATA, wCloseCallNumber, IData);
    wCloseCallNumber++;
    wordNum = wCloseCallNumber + '0';

#ifdef DEBUG_DLG
    if (DebugFlag & DEBUG_Recog)
#endif
    SpeechEnable((LPSTR) &wordNum);
}

```

```

/*-----
FUNCTION  _LOCAL int PhraseListInc(void)
DESCRIPTION Return close call list increment
PARAMETERS None.
RETURN   Close call list increment
*/
_LOCAL int PhraseListInc(void)
{
    RECT Rect;
    int ListSize;
    int CloseCallSize;

    CloseCallSize = wCloseCallNumber * (cyStatusText + 1);
    GetClientRect(hwndStatus, (LPRECT) &Rect);
    ListSize = Rect.bottom - iStatusSizeMin;
    if (ListSize < 0)
    {
        ListSize = 0;
    }
    return((ListSize >= CloseCallSize) ? 0 : CloseCallSize - ListSize);
}

```

```

/*-----
FUNCTION  BOOL PhraseListAdd(szStr, ContextEntry)
DESCRIPTION Add phrase to the phrase list.
PARAMETERS PSTR szStr    - Specifies pointer to the phrase.
            int ContextEntry - Specifies index in the context list.
RETURN    TRUE if success.
*/
BOOL PhraseListAdd(char * szStr, int ContextEntry)
{
    UINT wIdx;
    BOOL bWord = FALSE;

    if (szStr == NULL)
    {
        return(TRUE);
    }

    if (ContextEntry == -1)
    {
        /*
        ** Has no context link so look for one.
        */
        if (PhraseFind(szStr) != -1) return(TRUE);
    }
}

```

```

    }

#ifdef DEBUG_DLG
    if (DebugFlag & DEBUG_Recog)
#endif
        bWord = SpeechEnable(szStr);

    /*
    ** Now add it to the list.
    */
    wIdx = (UINT) SendMessage(hwndList, LB_ADDSTRING, 0, (DWORD)(LPSTR) szStr);
    if (wIdx == (UINT)LB_ERR)
    {
        return(FALSE);
    }
    SendMessage(hwndList, LB_SETITEMDATA, wIdx, MAKELONG(ContextEntry,
bWord));
    return(TRUE);
}

/*-----
FUNCTION  _LOCAL void PhraseListSetup(void)
DESCRIPTION Get the current set of words and give them to the recognizer.
PARAMETERS None.
RETURN    None.
*/
_LOCAL void PhraseListSetup(void)
{
    UINT wIdx;
    RECT rc;

    if (! hwndList)
        return;

    SendMessage(hwndList, WM_SETREDRAW, FALSE, 0);
    SendMessage(hwndList, LB_RESETCONTENT, 0, 0);

#ifdef DEBUG_DLG
    if (DebugFlag & DEBUG_Recog)
#endif
        SpeechDisableAll();           /* Disable all words. */

    ContextListAdd();                 /* Get context first. */

    if (wCloseCallInc)
    {
        /*
        ** Resize window to normal
        */
        GetWindowRect(hwndStatus, &rc);
    }
}

```

```

        rc.bottom -= wCloseCallInc;
        wCloseCallInc = 0;
        MoveWindow(
            hwndStatus,
            rc.left,
            rc.top,
            rc.right - rc.left,
            rc.bottom - rc.top,
            TRUE);
    }

    if (bCloseCallWas)
    {
        /*
        ** Include CloseCall information
        */
        wCloseCallNumber = 0;
        for (wIdx = 0; wIdx < vrState.nWords; PhraseListMove(vrState.word[wIdx ++]));

        if (! IsIconic(hwndStatus))
        {
            /*
            ** Should we resize PhraseList ?
            */
            wCloseCallInc = PhraseListInc();
            if (wCloseCallInc) {
                GetWindowRect(hwndStatus, &rc);
                MoveWindow(
                    hwndStatus,
                    rc.left,
                    rc.top,
                    rc.right - rc.left,
                    rc.bottom - rc.top + wCloseCallInc,
                    TRUE);
            }
        }

        }

    SendMessage(hwndList, WM_SETREDRAW, TRUE, 0);
}

/*-----
FUNCTION  _LOCAL void PhraseDrawItem(LPDRAWITEMSTRUCT lpd)
DESCRIPTION Draw item routine for the status item.
PARAMETERS LPDRAWITEMSTRUCT lpd - Specifies pointer to the DRAWITEMSTUCT.
RETURN    None.
*/
_LOCAL void PhraseDrawItem(LPDRAWITEMSTRUCT lpd)
{

```



```

HBRUSH hBrush;
int  iBkColor;
int  iTxColor;
char  szWord[2 * MAX_SYMBOL_LENGTH + 50];

if (lpd->itemID == -1)
    return;

if ((lpd->itemState & ODS_SELECTED) && (lpd->itemState & ODS_FOCUS))
{
    iBkColor = COLOR_HIGHLIGHT;
    iTxColor = COLOR_HIGHLIGHTTEXT;
}
else
{
    iBkColor = COLOR_WINDOW;
    iTxColor = COLOR_WINDOWTEXT;
}

SetTextColor(lpd->hDC, GetSysColor(iTxColor));
SetBkColor( lpd->hDC, GetSysColor(iBkColor));

hBrush = CreateSolidBrush(GetSysColor(iBkColor));
FillRect(lpd->hDC, (LPRECT)&(lpd->rcItem), hBrush);
DeleteObject(hBrush);

/*
** Now draw the text.
*/
SendMessage(hwndList, LB_GETTEXT, lpd->itemID, (LONG)(LPSTR)szWord);
if (bCloseCallWas && lpd->itemID < wCloseCallNumber)
{
    PaintBitmap(
        lpd->hDC, lpd->rcItem.left, lpd->rcItem.top,
        BMP_SIZE, BMP_SIZE,
        hbmpAnd, hbmpPaint, lpd->itemID * BMP_SIZE, 0);
    TextOut(lpd->hDC, lpd->rcItem.left + BMP_SIZE, lpd->rcItem.top, szWord,
        lstrlen(szWord));
}
else
{
    if (!HIWORD(SendMessage(hwndList, LB_GETITEMDATA, lpd->itemID, 0L)))
    {
        SetTextColor(lpd->hDC, GetSysColor(COLOR_GRAYTEXT));
    }

#ifdef DEBUG_DLG
    if (DebugFlag & DEBUG_ConFull)
    {
        TabbedTextOut(lpd->hDC, lpd->rcItem.left, lpd->rcItem.top,
            szWord, lstrlen(szWord), 2, ContextTabs, lpd->rcItem.left);
    }
    else
#endif
        TextOut(lpd->hDC, lpd->rcItem.left, lpd->rcItem.top, szWord, lstrlen(szWord));
}

```

```

    }
}

/*-----
FUNCTION  _LOCAL void PhraseExec(widx)
DESCRIPTION Execute the links associated with the phrase.
PARAMETERS  UINT widx - Specifies index of the phrase in the listbox.
RETURN     None.
*/
_LOCAL void PhraseExec(UINT widx)
{
    if (widx != (UINT)LB_ERR)
    {
        StatusChange();
        /*
        ** Activate the context link macro. If it has one.
        */
        ContextListSelect(LOWORD(SendMessage(hwndList, LB_GETITEMDATA,
widx, NULL)));
    }
}

/*-----
FUNCTION  void CALLBACK PhraseTimerProc(hwnd, msg, idTimer, dwTime)
DESCRIPTION An application-defined callback function that
            processes WM_TIMER messages.
            Look for context change
PARAMETERS  HWND hwnd          - Identifies the window associated with the timer.
            UINT msg           - Specifies the WM_TIMER message.
            UINT idTimer       - Specifies the timer's identifier.
            DWORD dwTime       - Specifies the current system time.
RETURN     None.
*/
void CALLBACK PhraseTimerProc(HWND hwnd, UINT wMsg, UINT idTimer, DWORD dwTime)
{
    static BOOL Active = FALSE;

    if (!Active)
    {
        Active = TRUE;
        if (ContextCheck(FALSE))
        {
            bCloseCallWas = FALSE;

```



```

int sfNew = UserGetFlags();
RECT rc;
int yInc = 0;

iStatusSizeMin = 0;
if (sfNew & PREF_Volume)
    iStatusSizeMin += 4 + cyStatusText;
if (sfNew & PREF_Confid)
    iStatusSizeMin += 4 + cyStatusText;
iStatusSizeMin = max(iStatusSizeMin, 6 + GetSystemMetrics(SM_CYICON));
GetClientRect(hwnd, &rc);
if (rc.bottom < iStatusSizeMin)
    yInc = iStatusSizeMin - rc.bottom;
GetWindowRect(hwnd, &rc);
MoveWindow(
    hwnd,
    rc.left,
    rc.top,
    rc.right - rc.left,
    rc.bottom - rc.top + yInc,
    TRUE);
SendMessage(hwnd, WM_SIZE, 0, 0L);
InvalidateRect(hwnd, NULL, TRUE); /* rebuild if resized or not */
ContextCheck(TRUE);
PhraseListSetup();          /* rebuild the current vocab list. */

```

```

}

```

```

/*-----

```

```

FUNCTION _LOCAL void SelectOurFont()

```

```

DESCRIPTION Select font for phrase listbox.

```

```

PARAMETERS None.

```

```

RETURN None.

```

```

*/

```

```

_LOCAL void SelectOurFont()

```

```

{

```

```

HDC      hDC;
TEXTMETRIC tm;
HFONT hFontNew;

```

```

hFontNew = UserGetFont();

```

```

hDC = CreateIC((LPSTR)"DISPLAY", NULL, NULL, NULL);
SelectObject(hDC, hFontNew);
GetTextMetrics(hDC, &tm);

```

```

SendMessage(hwndList, WM_SETFONT, hFontNew, 0L);

```

```

        SendMessage(hwndList, LB_SETITEMHEIGHT, 0, MAKELONG(max(tm.tmHeight,
BMP_SIZE), 0));
        cxStatusText = tm.tmAveCharWidth;
        cyStatusText = tm.tmHeight;
        if (hFontCur != NULL)
            DeleteObject(hFontCur);
        hFontCur = hFontNew;
        DeleteDC(hDC);
    }

```

```

/*-----

```

```

FUNCTION    BOOL CALLBACK StatusWndProc(hwnd, wMsg, wParam, lParam)

```

```

DESCRIPTION Window Proc VoiceStatus class.

```

```

    The form of the status window is follows:

```

```

        Title bar = System menu icon, last word, w/ current

```

```

        Confidence

```

```

        Volume

```

```

        Current options list box.

```

```

PARAMETERS HWND hwnd - Specifies the handle of the window

```

```

        UINT wMsg - Specifies the message

```

```

        WORD wParam - Specifies 16 bits of additional
        message-dependent information

```

```

        LONG lParam - Specifies 16 bits of additional
        message-dependent information

```

```

RETURN      Depend upon the message.

```

```

*/

```

```

long FAR PASCAL StatusWndProc(HWND hwnd, UINT wMsg, WORD wParam, LONG lParam)

```

```

{

```

```

    static WORD  wMenuCmd = NULL;

```

```

    static DWORD dwMenuBits = NULL;

```

```

    static BOOL  bRecogReady = FALSE;

```

```

    switch (wMsg)
    {

```

```

        case WM_CREATE:

```

```

        {

```

```

            /* Install System

```

```

            */

```

```

            LPCREATESTRUCT lpcs = (LPCREATESTRUCT) lParam;

```

```

            /* Create the list of available words for the user.

```

```

            */

```

```

            hwndList = CreateWindow(

```

```

                "LISTBOX",

```

```

                NULL,

```

```

                WS_CHILD | WS_VISIBLE | WS_BORDER |

```

```

                WS_HSCROLL | LBS_NOINTEGRALHEIGHT |

```

```

                LBS_NOTIFY | LBS_OWNERDRAWFIXED |

```

```

                LBS_HASSTRINGS | LBS_WANTKEYBOARDINPUT,

```

```

                iStatusSizeMin,

```

```

        0,
        lpcs->cx - iStatusSizeMin,
        lpcs->cy,
        hwnd,
        IDLIST_PHRASE,
        VChInst,
        (LPSTR) NULL);
    if (hwndList == NULL)
    {
        return(-1);
    }

    /* Hook message queue
    */
    HookInstall(TRUE);

    /* Start DDE with Program Manager
    */
    ShellDdeInit(&VCTalk);

    /* Install help hook (F1 in dialogs and menu)
    */
    HelpHookInit();

    /* The window gets created, so do the one time stuff.
    */
    hicoMain = LoadIcon(VChInst, MAKEINTRESOURCE(ICO_MAIN));
    hicoStat = LoadIcon(VChInst, MAKEINTRESOURCE(ICO_STAT));
    hbmpPaint = LoadBitmap(VChInst,
MAKEINTRESOURCE(BMP_CLCALL));
    hbmpAnd = CreateAndBitmap(hbmpPaint);

#ifdef DEBUG_DLG
    /* Update system menu
    */
    {
        char szWork[MAXSTRING + 1];
        HMENU hMenu = GetSystemMenu(hwnd, FALSE);;

        AppendMenu(hMenu, MF_SEPARATOR, 0, 0);
        LoadString(VChInst, IDS_DEBUG, (LPSTR)szWork,
MAXSTRING);
        AppendMenu(hMenu, MF_STRING, IDM_SYSDEBUG,
(LPSTR)szWork);
        DrawMenuBar(hwnd);
    }
#endif

    /* Set prefs
    */
    SelectOurFont();
    StatusSetPref(hwnd);

    /* Status is owner of the speech channel
    */
    SpeechOwner(hwnd);

```

```

        /* Set the initial values to the prase list.
        */
        PhraseListSetup();

        bRecogReady = TRUE;

        /* Do not put break here.
        ** We change user from void to current
        */
    }

    case VCM_USERCHANGED:
    {
        RECT rc;
        HCURSOR hcur;
        HWND hwndEdit;

        hcur = SetCursor(LoadCursor(NULL, IDC_WAIT));

        /* Set Status placement
        */
        UserGetWinRect(szStatusClass, &rc);
        MoveWindow(
            hwnd,
            rc.left,
            rc.top,
            rc.right - rc.left,
            rc.bottom - rc.top,
            TRUE);
        StopTimer();
        bRecogReady = FALSE;

        /* Load voice file
        */
#ifdef DEBUG_DLG
        if (DebugFlag & DEBUG_Recog)
#endif
            SpeechUserChange();

        /* Load Language
        */
        hwndEdit = FindWindow(szFrameClass, NULL);
        if (hwndEdit != NULL)
        {
            /* Load from the editor
            */
            ContextNewLang((LPLANG)SendMessage(hwndEdit,
iEditChangeMsg, 0, 0L));
        }
        else
        {
            /* Load from the file
            */
            ContextNewLang(NULL);
        }
    }

```

```

bRecogReady = TRUE;
StartTimer();
SetCursor(hcur);
PhraseListSetup();
break;

```

```

}

```

```

case WM_MENUSELECT:

```

```

    /* Keep menu selection for help
    */

```

```

    dwMenuBits=lParam;
    wMenuCmd=wParam;
    goto defmsg;

```

```

case VCM_HELP:

```

```

    if (!(LOWORD(dwMenuBits) & MF_POPUP))
    {

```

```

        if (!(LOWORD(dwMenuBits) & MF_SYSMENU))
        {

```

```

            /* Menu help
            */

```

```

            Help(hwnd, HELP_VCMenuPrefs + wMenuCmd -

```

```

MENU_STATUS);

```

```

        }
        else

```

```

        {

```

```

            /* System menu help
            */

```

```

            Help(hwnd, HELP_SysMenu);

```

```

        }

```

```

    }
    else

```

```

    {

```

```

        /* General help
        */

```

```

        Help(hwnd, HELP_Status);

```

```

    }

```

```

    break;

```

```

case VCM_SPEECH:

```

```

{

```

```

    /*

```

```

    ** Speech available

```

```

    */

```

```

    UINT wIdx;

```

```

    UINT wUtt;

```

```

    if (bRecogReady && !bPause)

```

```

    {

```

```

        bRecogReady = FALSE;

```

```

        StopTimer();

```

```

        wUtt = SpeechRecog(&vrState);

```

```

        /* Check Close Call list first.

```



```

        */
        if (wUtt != 0)
        {
            if (vrState.confidence >= UserGetConfidence()) {
                if (bCloseCallWas) {
                    wIdx = CloseCallFind(vrState.word[0]);
                    if (wIdx != (UINT)LB_ERR) {
                        SendMessage(hwndList,
                            LB_GETTEXT, wIdx, (LONG)(LPSTR)(vrState.word[0]));
                        if ( UserGetFlags() &
                            PREF_Adapt) {
                            SpeechAdapt(vrState.w
                                ord[0], wUttCloseCall);
                        }
                    }
                    else {
                        wIdx =
                            PhraseFind(vrState.word[0]);
                    }
                }
                else {
                    wIdx = PhraseFind(vrState.word[0]);
                }
                bCloseCallWas = FALSE;
                SpeechErase();

                /* A word was recognized correctly.
                */
                PhraseExec(wIdx);
            }
            else {
                /* Setup Close Call list
                */
                bCloseCallWas = TRUE;
                wUttCloseCall = wUtt;
                StatusChange();
                PhraseListSetup();
            }
        }
        StartTimer();
        bRecogReady = TRUE;
    }
    break;
}

case VCM_TRAIN:
{
    /* Word was trained
    */
    UINT wIdx;
    LONG lData;
    RECT rc;

    wIdx = PhraseFind((PSTR)lParam);
    if (wIdx != (UINT)LB_ERR) {
        lData = SendMessage(hwndList, LB_GETITEMDATA, wIdx, 0L);
    }
}

```

```

        if (!HIWORD(IData)) {
            SendMessage(hwndList, LB_SETITEMDATA, wldx,
                MAKELONG(LOWORD(IData), TRUE));
            SendMessage(hwndList, LB_GETITEMRECT, wldx,
                (LONG)(LPRECT)&rc);
            InvalidateRect(hwndList, &rc, TRUE);
        }
    }
    break;
}

case WM_PAINT:
{
    /* A repaint instruction has been given.
    */
    HDC      hDC;
    PAINTSTRUCT ps;
    HICON     hIcon;

    hDC = BeginPaint(hwndStatus, (LPPAINTSTRUCT)&ps);
    if (!IsIconic(hwndStatus))
    {
        /* Draw iconic window
        */
        hIcon = (bPause) ? hicoMain : hicoStat;
        DrawIcon(hDC, 0, 0, hIcon);
    }
    else
    {
        /* Create the volume and confidence boxes.
        */
        StatusBars(hDC);
    }
    EndPaint(hwndStatus, (LPPAINTSTRUCT)&ps);
    break;
}

case WM_SIZE:
{
    /* Move the phrase list.
    */
    RECT rc;

    GetClientRect(hwnd, &rc);
    MoveWindow(
        hwndList,
        rc.left,
        rc.top + iStatusSizeMin,
        rc.right - rc.left + 1,
        rc.bottom - rc.top - iStatusSizeMin + 1,
        TRUE);
    break;
}

case WM_GETMINMAXINFO:
{

```

```

MINMAXINFO FAR * lpmmi = (MINMAXINFO FAR *) lParam;
RECT rc;

memset(&rc, 0, sizeof(rc));
rc.bottom = iStatusSizeMin;
AdjustWindowRect(&rc, WS_OVERLAPPEDWINDOW, TRUE);

lpmmi->ptMinTrackSize.x = MAX_SYMBOL_LENGTH * cxStatusText;
lpmmi->ptMinTrackSize.y = rc.bottom - rc.top + wCloseCallInc;
break;
}

```

```

case WM_SETFOCUS:
    /* We just got the focus.
    */
    SetFocus(hwndList);          /* Give it to the list box. */
    break;

```

```

case WM_QUERYDRAGICON:
    /* A repaint instruction has been given.
    */
    return(bPause ? hicoMain : hicoStat);

```

```

case WM_DRAWITEM:
    /* The system listbox wants us to draw the item.
    ** DRAWITEMSTRUCT
    */
    PhraseDrawItem((LPDRAWITEMSTRUCT) lParam);
    break;

```

```

#ifdef DEBUG_DLG
    case WM_SYSCOMMAND:
        if ((wParam & 0xFFF0) == IDM_SYSDEBUG)
        {
            /* Bring up the Debug dialog box.
            */
            DialogBox(VChInst, MAKEINTRESOURCE(DLG_DEBUG),
hwnd, DebugDlgProc);

            /* Rebuild phrase list
            */
            PhraseListSetup();
        }
        else
        {
            goto defmsg;
        }
        break;

```

```

#endif

```

```

case WM_COMMAND:
    switch (wParam)
    {
        case IDM_PREFS:
            /* Bring up the User Prefereces dialog box.

```

```

        */
        if(UserPref(hwnd))
        {
            SelectOurFont();
        }
        StatusSetPref(hwnd);
        break;

case IDM_TRAIN:
    /* Bring up the Vocabulary Training dialog box.
    */
    SendMessage(hwnd, WM_COMMAND,
IDLIST_PHRASE, MAKELONG(0, LBN_DBLCLK));
    break;

case IDM_PAUSE:
{
    /* Pause on/off.
    */
    char szTitle[MAXSTRING + 1];

    bPause = ! bPause;
    CheckMenuItem(GetMenu(hwnd), IDM_PAUSE,
        MF_BYCOMMAND | (bPause ? MF_CHECKED
: MF_UNCHECKED));

    LoadString(VChInst, IDS_TITLE, (LPSTR)szTitle,
sizeof(szTitle));

    SetWindowText(hwnd, (LPSTR)szTitle);
    InvalidateRect(hwnd, NULL, TRUE) ;
    break;
}

case IDM_EDIT:
{
    /*
    ** Bring up the Language Editor
    */
    char szVeFile[MAXFILENAME + 1];

    IniGetVeFile(szVeFile);
    WinExec(szVeFile, SW_SHOW);
    break;
}

case IDM_EXIT:
    /* Exit now
    */
    SendMessage(hwnd, WM_CLOSE, 0, 0L);
    break;

case IDM_HELPCONTENT:
    /* Bring up the Help
    */
    Help(hwnd, HELP_Status);
    break;

```

09852049 050901  
T06050" 64025860

```

case IDM_HELPSEARCH:
    /* Bring up the Help Search
    */
    Help(hwnd, HELP_Search);
    break;

case IDM_HELPONHELP:
    /* Bring up the HelpOnHelp
    */
    Help(hwnd, HELP_OnHelp);
    break;

case IDM_ABOUT:
    /* Bring up the About.. dialog box.
    */
    About(hwnd);
    break;

case IDLIST_PHRASE:
    switch (HIWORD(IParam)) {
        case LBN_DBLCLK:

#ifdef DEBUG_DLG
            if (DebugFlag & DEBUG_Force) {
                /* Execute command
                */
                char * Ptr;
                UINT widx =

                (UINT)SendMessage(hwndList, LB_GETCURSEL, 0, 0);

                vrState.confidence = 100;
                vrState.amplitude = 0;
                SendMessage(hwndList,

                LB_GETTEXT, widx, (LONG)(LPSTR)(vrState.word[0]));

                DEBUG_ContFull)

                information

                vrState.word[0]; *Ptr; Ptr++)

                '\0';

                if (*Ptr == '\t')
                {
                    *Ptr =

                    break;
                }
            }
            PhraseExec(widx);
            break;
        }
    }

#endif

/* Train command
*/

```

```

                                TrainExec(TRUE,
(UINT)SendMessage(hwndList, LB_GETCURSEL, 0, 0), hwndList);
                                break;
                                case LBN_SETFOCUS:
                                    /* We just got focus. clear previous
inputs.
                                    */
                                    break;
                                default :
                                    goto defmsg;
                                }
                                break;

                                default:
                                    goto defmsg;

                                }
                                break;

case WM_QUERYENDSESSION:
{
    WINDOWPLACEMENT wndpl;
    HWND hwndEdit;

    if (wParam == 2)
    {
        /* We don't quit, just hange user
        */
        hwndEdit = FindWindow(szFrameClass, NULL);
        if (hwndEdit != NULL)
        {
            Error(ERREditExist);
            ShowWindow(hwndEdit, SW_SHOWNORMAL);
            SetFocus(hwndEdit);
            break;
        }
    }

    /* Save users settings
    */
    wndpl.length = sizeof(wndpl);
    GetWindowPlacement(hwnd, &wndpl);
    UserSetWinRect(szStatusClass, &(wndpl.rcNormalPosition));
    goto defmsg;
}

case WM_CLOSE:
    /* Ask permission before quit
    */
    if (CallTaskWindows(TRUE, WM_QUERYENDSESSION, TRUE, 0L))
    {
        CallTaskWindows(FALSE, WM_DESTROY, 0, 0L);
    }
    break;

case WM_DESTROY :

```

```

{
    /* Free resores
    */
    HCURSOR hcur;

    hcur = SetCursor(LoadCursor(NULL, IDC_WAIT));
    StopTimer();
    DestroyIcon(hicoStat);
    DestroyIcon(hicoMain);
    DeleteObject(hbmpPaint);
    DeleteObject(hbmpAnd);
    DeleteObject(hFontCur);

    /* Free speech system
    */
#ifdef DEBUG_DLG
    if (DebugFlag & DEBUG_Recog)
#endif
        SpeechFree();

    /* Unhook message queue
    */
    HookInstall(FALSE);
    HookFreeJournal();

    /* Close help if was opened
    */
    Help(hwnd, HELP_Quit);

    /* Stop DDE
    */
    ShellDdeExit(&VCTalk);

    /* Unhook help hook
    */
    HelpHookExit();

    /* Save the user file.
    */
    UserExit();
    SetCursor(hcur);

    /* Kill the task and other windows.
    */
    PostQuitMessage(0);
    break;
}
default:
    if (wMsg == iEditChangeMsg)
    {
        /* Changes in Editor saved
        ** We need to update language
        */
        HCURSOR hcur;

```

```

        hcur = SetCursor(LoadCursor(NULL, IDC_WAIT));
        StopTimer();
        bRecogReady = FALSE;

```

```

        /* Load Language
        */
        ContextNewLang((LPLANG)IParam);
        bRecogReady = TRUE;
        StartTimer();
        SetCursor(hcur);
        break;
    }

```

```

    defmsg:
        return DefWindowProc(hwnd, wMsg, wParam, lParam);
}

return (NULL);

```

```

/*-----
FUNCTION  BOOL StatusInit(BOOL bNew)

DESCRIPTION

PARAMETERS

RETURN

*/
BOOL StatusInit(BOOL bNew)
{

```

```

    WNDCLASS wc;
    char  szTitle[MAXSTRING + 1];
    RECT  rc;
    HWND  hwnd;

```

```

    if (bNew)
    {
        UserInit();

```

```

        /* To reload file
        */
        iEditChangeMsg = RegisterWindowMessage(szFrameClass);

```

```

        /* Register the window class.
        */
        memset(&wc, 0, sizeof(wc));          /* zero structure to start. */

```

```

        wc.style      = CS_DBLCLKS | CS_HREDRAW | CS_VREDRAW ;
        wc.lpfnWndProc = (WNDPROC)StatusWndProc;
        wc.hInstance  = VChInst;              /* task owner. */
        wc.hCursor    = LoadCursor(NULL, IDC_ARROW) ;

```

"C:\WINDOWS\SYSTEM32\cmd.exe" /c dir



```

wc.hbrBackground = COLOR_BTNFACE + 1;
wc.lpszClassName = (LPSTR) szStatusClass;
wc.lpszMenuName = MAKEINTRESOURCE(MENU_STATUS);

if (! RegisterClass(&wc))
    return(FALSE);

hAccTableStatus = LoadAccelerators(VChInst,
MAKEINTRESOURCE(ATBL_STATUS));

/* Create Status Window
*/
UserGetWinRect(szStatusClass, &rc);
LoadString(VChInst, IDS_TITLE, (LPSTR)szTitle, sizeof(szTitle) - 1);

hwndStatus = CreateWindowEx(
    WS_EX_TOPMOST,
    szStatusClass,
    (LPSTR)szTitle,
    WS_OVERLAPPEDWINDOW & (~WS_MAXIMIZEBOX),
    rc.left,
    rc.top,
    rc.right - rc.left,
    rc.bottom - rc.top,
    NULL,
    NULL,
    VChInst,
    (LPSTR) NULL);

if (! hwndStatus)
    return(FALSE);

/* Send timer message to update context.
** every 1/2 of a second or so.
*/
StartTimer();
ShowWindow(hwndStatus, SW_SHOWNORMAL);

/* Install recognition system
*/
#ifdef DEBUG_DLG
    if (DebugFlag & DEBUG_Recog)
        SpeechInit();

        PhraseListSetup();
}
else
{
    /* Only one instance of Voice Control should be present
    */
    hwnd = FindWindow(szStatusClass, NULL);
    if (hwnd)
    {
        /* This should always be true !?

```



| Table 1. Demographic characteristics of the study population |            |
|--|------------|
| Age (years)  | 65.0 ± 1.5 |
| Gender (male/female)   | 10/10      |
| Education (years)  | 12.0 ± 1.0 |
| Occupation (white/blue)                                      | 10/10      |
| Marital status (married/divorced/widowed)                    | 10/10/0    |
| Smoking status (smoker/nonsmoker)                            | 10/10      |
| Alcohol consumption (yes/no)                                 | 10/10      |
| Comorbidities (hypertension/diabetes/cholesterol)            | 10/10/10   |
| Medication (antihypertensive/antidiabetic/anticholesterol)   | 10/10/10   |
| Physical activity (yes/no)                                   | 10/10      |
| Stress level (low/high)                                      | 10/10      |
| Sleep quality (good/poor)                                    | 10/10      |
| Depression score (0-10)                                      | 5.0 ± 1.0  |
| Overall health (good/fair/poor)                              | 10/10/0    |

This invention relates to voice controlled computer interfaces.

5

10

## 15

20

35

actions to be taken by said operating system, and converting other voiced utterances into commands which carry associated text strings to be used as part of text being processed in an application program running under the operating system.

5 In general, in another aspect, the invention features generating a table for aiding the conversion of voiced utterances to commands for use in controlling an operating system of a computer to achieve desired actions in an application program running under the operating system, the application program  
10 including menus and control buttons; the instruction sequence of the application program is parsed to identify menu entries and control buttons, and an entry is included in the table for each menu entry and control button found in the application program, each entry in the table containing a command corresponding to the  
15 menu entry or control button.

In general, in another aspect, the invention features enabling a user to create an instance in a formal language of the kind which has a strictly defined syntax; a graphically displayed list of entries are expressed in a natural language and do not comply with  
20 the syntax, the user is permitted to point to an entry on the list, and the instance corresponding to the identified entry in the list is automatically generated in response to the pointing.

The invention enables a user to easily control the graphical interface of a computer. Any actions that the operating system can  
25 be commanded to take can be commanded by voiced utterances. The commands may include commands that are normally entered through the keyboard as well as commands normally entered through a mouse or any other input device. The user may switch back and forth between voiced utterances that correspond to commands for actions  
30 to be taken and voiced utterances that correspond to text strings to be used in an application program without giving any indication that the switch has been made. Any application may be made susceptible to a voice interface by automatically parsing the application instruction sequence for menus and control buttons that  
35 control the application.

09852049-050901  
T06050" 6405860

Other advantages and features will become apparent from the following description of the preferred embodiment and from the claims.

#### Description of the Preferred Embodiment

5 We first briefly describe the drawings.

Fig. 1 is a functional block diagram of a Macintosh computer served by a Voice Navigator voice controlled interface system.

Fig. 2A is a functional block diagram of a Language Maker system for creating word lists for use with the Voice Navigator interface of Fig. 1.

Fig. 2B depicts the format of the voice files and word lists used with the Voice Navigator interface.

Fig. 3 is an organizational block diagram of the Voice Navigator interface system.

15 Fig. 4 is a flow diagram of the Language Maker main event loop.

Fig. 5 is a flow diagram of the Run Edit module.

Fig. 6 is a flow diagram of the Record Actions submodule.

Fig. 7 is a flow diagram of the Run Modal module.

20 Fig. 8 is a flow diagram of the In Button? routine.

Fig. 9 is a flow diagram of the Event Handler module.

Fig. 10 is a flow diagram of the Do My Menu module.

Figs. 11A through 11I are flow diagrams of the Language Maker menu submodules.

25 Fig. 12 is a flow diagram of the Write Production module.

Fig. 13 is a flow diagram of the Write Terminal submodule.

Fig. 14 is a flow diagram of the Voice Control main driver loop.

Fig. 15 is a flow diagram of the Process Input module.

30 Fig. 16 is a flow diagram of the Recognize submodule.

Fig. 17 is a flow diagram of the Process Voice Control Commands routine.

Fig. 18 is a flow diagram of the ProcessQ module.

Fig. 19 is a flow diagram of the Get Next submodule.

35 Fig. 20 is a chart of the command handlers.

Figs. 21A through 21G are flow diagrams of the command handlers.

Fig. 22 is a flow diagram of the Post Mouse routine.

Fig. 23 is a flow diagram of the Set Mouse Down routine.

5 Figs. 24 and 25 illustrate the screen displays of Voice Control.

Figs. 26 through 29 illustrate the screen displays of Language Maker.

Fig. 30 is a listing of a language file.

## 10 System Overview

Referring to Fig. 1, in an Apple Macintosh computer 100, a Macintosh operating system 132 provides a graphical interactive user interface by processing events received from a mouse 134 and a keyboard 136 and by providing displays including icons, windows, and menus on a display device 138. Operating system 132 provides an environment in which application programs such as MacWrite 139, desktop utilities such as Calculator 137, and a wide variety of other programs can be run.

20 The operating system 132 also receives events from the Voice Navigator voice controlled computer interface 102 to enable the user to control the computer by voiced utterances. For this purpose, the user speaks into a microphone 114 connected via a Voice Navigator box 112 to the SCSI (Small Computer Systems Interface) port of the computer 100. The Voice Navigator box 112 digitizes and processes analog audio signals received from a microphone 114, and transmits processed digitized audio signals to the Macintosh SCSI port. The Voice Navigator box includes an analog-to-digital converter (A/D) for digitizing the audio signal, a DSP (Digital Signal Processing) chip for compressing the resulting digital samples, and protocol interface hardware which configures the digital samples to obey the SCSI protocols.

30 Recognizer Software 120 (available from Dragon Systems, Newton, MA) runs under the Macintosh operating system, and is controlled by internal commands 123 received from Voice Control driver 128 (which also operates under the Macintosh operating

35

system). One possible algorithm for implementing Recognizer Software 120 is disclosed by Baker et al, in US Patent 4,783,803, incorporated by reference herein. Recognizer Software 120 processes the incoming compressed, digitized audio, and compares each utterance of the user to prestored utterance macros. If the user utterance matches a prestored utterance macro, the utterance is recognized, and a command string 121 corresponding to the recognized utterance is delivered to a text buffer 126. Command strings 121 delivered from the Recognizer Software represent commands to be issued to the Macintosh operating system (e.g., menu selections to be made or text to be displayed), or internal commands 123 to be issued by the Voice Control driver.

During recognition, the Recognizer Software 120 compares the incoming samples of an utterance with macros in a voice file 122. (The system requires the user to space apart his utterances briefly so that the system can recognize when each utterance ends.) The voice file macros are created by a "training" process, described below. If a match is found (as judged by the recognition algorithm of the Recognizer Software 120), a Voice Control command string from a word list 124 (which has been directly associated with voice file 122) is fetched and sent to text buffer 126.

The command strings in text buffer 126 are relayed to Voice Control driver 128, which drives a Voice Control interpreter 130 in response to the strings.

A command string 121 may indicate an internal command 123, such as a command to the Recognizer Software to "learn" new voice file macros, or to adjust the sensitivity of the recognition algorithm. In this case, Voice Control interpreter 130 sends the appropriate internal command 123 to the Recognizer Software 120. In other cases, the command string may represent an operating system manipulation, such as a mouse movement. In this case, Voice Control interpreter 130 produces the appropriate action by interacting with the Macintosh operating system 132.

Each application or desktop accessory is associated with a word list 124 and a corresponding voice file 122; these are loaded

T0650" 4025880

by the Recognition Software when the application or desktop accessory is opened.

The voice files are generated by the Recognizer Software 120 in its "learn" mode, under the control of internal commands from the Voice Control driver 128.

The word lists are generated by the Language Maker desktop accessory 140, which creates "languages" of utterance names and associated Voice Control command strings, and converts the languages into the word lists. Voice Control command strings are strings such as "ESC", "TEXT", "@MENU(font,2)", and belong to a Voice Control command set, the syntax of which will be described later and is set forth in Appendix A.

The Voice Control and Language Maker software includes about 30,000 lines of code, most of which is written in the C language, the remainder being written in assembly language. A listing of the Voice Control and Language Maker software is provided in microfiche as appendix C. The Voice Control software will operate on a Macintosh Plus or later models, configured with a minimum of 1 Mbyte RAM (2 Mbyte for HyperCard and other large applications), a Hard Disk, and with Macintosh operating system version 6.01 or later.

In order to understand the interaction of the Voice Control interpreter 130 and the operating system, note that Macintosh operating system 132 is "event driven". The operating system maintains an event queue (not shown); input devices such as the mouse 134 or the keyboard 136 "post" events to this queue to cause the operating system to, for example, create the appropriate text entry, or trigger a mouse movement. The operating system 132 then, for example, passes messages to Macintosh applications (such as MacWrite 139) or to desktop accessories (such as Calculator 137) indicating events on the queues (if any). In one mode of operation, Voice Control interpreter 130 likewise controls the operating system (and hence the applications and desktop accessories which are currently running) by posting events to the operating system queues. The events posted by the Voice Control



interpreter typically correspond to mouse activity or to keyboard keystrokes, or both, depending upon the voice commands. Thus, the Voice Navigator system 102 provides an additional user interface. In some cases, the "voice" events may comprise text strings to be displayed or included with text being processed by the application program.

At any time during the operation of the Voice Navigator system, the Recognizer Software 120 may be trained to recognize an utterance of a particular user and to associate a corresponding text string with each utterance. In this mode, the Recognizer Software 120 displays to the user a menu of the utterance names (such as "file", "page down") which are to be recognized. These names, and the corresponding Voice Control command strings (indicating the appropriate actions) appear in a current word list 124. The user designates the utterance name of interest and then is prompted to speak the utterance corresponding to that name. For example, if the utterance name is "file", the user might utter "FILE" or "PLEASE FILE". The digitized samples from the Voice Navigator box 112 corresponding to that utterance are then used by the Recognizer Software 120 to create a "macro" representing the utterance, which is stored in the voice file 122 and subsequently associated with the utterance name in the word list 124. Ordinarily, the utterance is repeated more than once, in order to create a macro for the utterance that accommodates variation in a particular speaker's voice.

The meaning of the spoken utterance need not correspond to the utterance name, and the text of the utterance name need not correspond to the Voice Control command strings stored in the word list. For example, the user may wish a command string that causes the operating system to save a file to have the utterance name "save file"; the associated command string may be "@MENU(file,2)"; and the utterance that the user trains for this utterance name may be the spoken phrase "immortalize". The Recognizer Software and Voice Control cause that utterance, name, and command string to be properly associated in the voice file and word list 124.

Referring to Fig. 2A, the word lists 124 used by the Voice Navigator are created by the Language Maker desk accessory 140 running under the operating system. Each word list 124 is hierarchical, that is, some utterance names in the list link to sub-lists of other utterance names. Only the list of utterance names at a currently active level of the hierarchy can be recognized. (In the current embodiment, the number of utterance names at each level of the hierarchy can be as large as 1000.) In the operation of Voice Control, some utterances, such as "file", may summon the file menu on the screen, and link to a subsequent list of utterance names at a lower hierarchical level. For example, the file menu may list subsequent commands such as "save", "open", or "save as", each associated with an utterance.

Language Maker enables the user to create a hierarchical language of utterance names and associated command strings, rearrange the hierarchy of the language, and add new utterance names. Then, when the language is in the form that the user desires, the language is converted to a word list 124. Because the hierarchy of the utterance names and command strings can be adjusted, when using the Voice Navigator system the user is not bound by the preset menu hierarchy of an application. For example, the user may want to create a "save" command at the top level of the utterance hierarchy that directly saves a file without first summoning the file menu. Also, the user may, for example, create a new utterance name "goodbye", that saves a file and exits all at once.

Each language created by Language Maker 140 also contains the command strings which represent the actions (e.g. clicking the mouse at a location, typing text on the screen) to be associated with utterances and utterance names. In order for the training of the Voice Navigator system to be more intuitive, the user does not specify the command strings to describe the actions he wishes to be associated with an utterance and utterance name. In fact, the user does not need to know about, and never sees, the command strings stored in the Language Maker language or the resulting word

list 124.

In a "record" mode, to associate a series of actions with an utterance name, the user simply performs the desired actions (such as typing the text at the keyboard, or clicking the mouse at a menu). The actions performed are converted into the appropriate command strings, and when the user turns off the record mode, the command strings are associated with the selected utterance name.

While using Language Maker, the user can cause the creation of a language by entering utterance names by typing the names at the keyboard 142, by using a "create default text" procedure 146 (to parse a text file on the clipboard, in which case one utterance name is created for each word in the text file, and the names all start at the same hierarchical level), or by using a "create default menus" procedure (to parse the executable code 144 for an application, and create a set of utterance names which equal the names of the commands in the menus of the application, in which case the initial hierarchy for the names is the same as the hierarchy of the menus in the application).

If the names are typed at the keyboard or created by parsing a text file, the names are initially associated with the keystrokes which, when typed at the keyboard, produce the name. Therefore, the name "text" would be initially be associated with the keystrokes t-e-x-t. If the names are created by parsing the executable code 144 for an application, then the names are initially associated with the command strings which execute the corresponding menu commands for the application. These initial command strings can be changed by simply selecting the utterance name to be changed and putting Language Maker into record mode.

The output of Language Maker is a language file 148. This file contains the utterance names and the corresponding command strings. The language file 148 is formatted for input to a VOCAL compiler 150 (available from Dragon Systems), which converts the language file into a word list 124 for use with the Recognition Software. The syntax of language files is specified in the Voice Navigator Developer's Reference Manual, provided as Appendix D, and

T0650" 64025890

Referring to Fig. 2B, a macro 147 of each learned utterance is stored in the voice file 122. A corresponding utterance name 149 and command string 151 are associated with one another and with the utterance and are stored in the word list 124. The word list 124 is created and modified by Language Maker 140, and the voice file 122 is created and modified by the Recognition Software 120 in its learn mode, under the control of the Voice Control driver 128.

The Voice Navigator system also includes the Recognition Software voice drivers 120 which include routines for utterance detection 164 and command execution 166. For utterance detection 164, the voice drivers periodically poll 168 the Voice Navigator hardware to determine if an utterance is being received by Voice Navigator box 152, based on the amplitude of the signal received by the microphone. When an utterance is detected 170, the voice drivers create a speech buffer of encoded digital samples (tokens) to be used by the command execution drivers 166. On command 166 from the Voice Control driver 128, the recognition drivers can learn new utterances by token-to-terminal conversion 174. The token is converted to a macro for the utterance, and stored as a terminal in a voice file 122 (Fig. 1).

Recognition and pattern matching 172 is also performed on command by the voice drivers. During recognition, a stored token of incoming digitized samples is compared with macros for the utterances in the current level of the recognition hierarchy. If a match is found, terminal to output conversion 176 is also performed, selecting the command string associated with the

recognized utterance from the word list 124 (Fig. 1). State management 178, such as changing of sensitivity controls, is also performed on command by the voice drivers.

5 The Voice Control driver 128 forms an interface 182 to the voice drivers 120 through control commands, an interface 184 to the Macintosh operating system 132 (Fig. 1) through event posting and operating system hooks, and an interface 186 to the user through display menus and prompts.

10 The interface 182 to the drivers allows Voice Control access to the Voice Driver command functions 166. This interface allows Voice Control to monitor 188 the status of the recognizer, for example to check for an utterance token in the utterance queue buffered 170 to the Macintosh. If there is an utterance, and if processor time is available, Voice Control issues command  
15 sdi\_recognize 190, calling the recognition and pattern match routine 172 in the voice drivers. In addition, the interface to the drivers may issue command sdi\_output 192 which controls the terminal to output conversion routine 176 in the voice drivers, converting a recognized utterance to an command string for use by  
20 Voice Control. The command string may indicate mouse or keystroke events to be posted to the operating system, or may indicate commands to Voice Control itself (e.g. enabling or disabling Voice Control).

25 From the user's perspective, Voice Control is simply a Macintosh driver with internal parameters, such as sensitivity, and internal commands, such as commands to learn new utterances. The actual processing which the user perceives as Voice Control may actually be performed by Voice Control, or by the Voice Drivers, depending upon the function. For example, the utterance  
30 learning procedures are performed by the Voice Drivers under the control of Voice Control.

The interface 184 to the Macintosh operating system allows Voice Control, where appropriate, to manipulate the operating system (e.g., by posting events or modifying event queues). The  
35 macro interpreter 194 takes the command strings delivered from the

voice drivers via the text buffer and interprets them to decide what actions to take. These commands may indicate text strings to be displayed on the display or mouse movements or menu selections to be executed.

5        In the interpretive execution of the command strings, Voice Control must manipulate the Macintosh event queues. This task is performed by OS event management 196. As discussed above, voice events may simulate events which are ordinarily associated with the keyboard or with the mouse. Keyboard events are handled by OS  
10    event management 196 directly. Mouse events are handled by mouse handler 198. Mouse events require an additional level of handling because mouse events can require operating system manipulation outside of the standard event post routines which are accomplished by the OS event management 196.

15        The main interface into the Macintosh operating system 132 is event based, and is used in the majority of the commands which are voice recognized and issued to the Macintosh. However, there are other "hooks" to the operating system state which are used to control parameters such as mouse placement and mouse motion. For  
20    example, as will be discussed later, pushing the mouse button down generates an event, however, keeping the mouse button pushed down and dragging the mouse across a menu requires the use of an operating system hook. For reference, the operating system hooks used by the Voice Navigator are listed in Appendix B.

25        The operating system hooks are implemented by the trap filters 200, which are filters used by Voice Control to force the Macintosh operating system to accept the controls implemented by OS event management 196 and mouse handler 198.

30        The Macintosh operating system traps are held in Macintosh read only memories (ROMs), and implement high level commands for controlling the system. Examples of these high level commands are: drawing a string onto the screen, window zooming, moving windows to the front and back of the screen, and polling the status of the mouse button. In order for the Voice Control driver to properly  
35    interface with the Macintosh operating system it must control these

operating system traps to generate the appropriate events.

To generate menu events, for example, Voice Control "seizes" the menu select trap (i.e. takes control of the trap from the operating system). Once Voice Control has seized the trap, application requests for menu selections are forwarded to Voice Control. In this way Voice Control is able to modify, where necessary, the operating system output to the program, thereby controlling the system behavior as desired.

The interface 186 to the user provides user control of the Voice Control operations. Prompts 202 display the name of each recognized utterance on the Macintosh screen so that the user may determine if the proper utterance has been recognized. On-line training 204 allows the user to access, at any time while using the Macintosh, the utterance names in the word list 124 currently in use. The user may see which utterance names have been trained and may retrain the utterance names in an on-line manner (these functions require Voice Control to use the Voice Driver interface, as discussed above). User options 206 provide selection of various Voice Control settings, such as the sensitivity and confidence level of the recognizer (i.e., the level of certainty required to decide that an utterance has been recognized). The optimal values for these parameters depend upon the microphone in use and the speaking voice of the user.

The interface 186 to the user does not operate via the Macintosh event interface. Rather, it is simply a recursive loop which controls the Recognition Software and the state of the Voice Control driver.

Language Maker 140 includes an application analyzer 210 and an event recorder 212. Application analyzer 210 parses the executable code of applications as discussed above, and produces suitable default utterance names and pre-programmed command strings. The application analyzer 210 includes a menu extraction procedure 214 which searches executable code to find text strings corresponding to menus. The application analyzer 210 also includes control identification procedures 216 for creating the command

strings corresponding to each menu item in an application.

The event recorder 212 is a driver for recording user commands and creating command strings for utterances. This allows the user to easily create and edit command strings as discussed above.

Types of events which may be entered into the event recorder include: text entry 218, mouse events 220 (such as clicking at a specified place on the screen), special events 222 which may be necessary to control a particular application, and voice events 224 which may be associated with operations of the Voice Control driver.

#### Language Maker

Referring to Fig. 4, the Language Maker main event loop 230 is similar in structure to main event loops used by other desk accessories in the Macintosh operating system. If a desk accessory is selected from the "Apple" menu, an "open" event is transmitted to the accessory. In general, if the application in which it resides quits or if the user quits it using its menus, a "close" event is transmitted to the accessory. Otherwise, the accessory is transmitted control events. The message parameter of a control event indicates the kind of event. As seen in Fig. 4, the Language Maker main event loop 230 begins with an analysis 232 of the event type.

If the event is an open event Language Maker tests 234 whether it is already opened. If Language Maker is already opened 236, the current language (i.e. the list of utterance names from the current word list) is displayed and Language Maker returns 237 to the operating system. If Language Maker is not open 238, it is initialized and then returns 239 to the operating system.

If the event is a close event, Language Maker prompts the user 240 to save the current language as a language file. If the user commands Language Maker to save the current language, the current language is converted by the Write Production module 242 to a language file, and then Language Maker exits 244. If the current language is not saved, Language Maker exits directly.

If the event is a control event 246, then the way in which



Language Maker responds to the event depends upon the mode that Language Maker is in, because Language Maker has a utility for recording events (i.e. the mouse movements and clicks or text entry that the user wishes to assign to an utterance), and must record events which do not involve the Language Maker window. However, when not recording, Language Maker should only respond to events in its window. Therefore, Language Maker may respond to events in one mode but not in another.

A control event 246 is forwarded to one of three branches 248, 250, 252. All menu events are forwarded to the accMenu branch 252. (Only menu events occurring in desk accessory menus will be forwarded to Language Maker.) All window events for the Language Maker window are forwarded to the accEvent branch 250. All other events received by Language Maker, which correspond to events for desktop accessories or applications other than Language Maker, initiate activity in the accRun branch 248, to enable recording of actions.

In the accRun branch 248, events are recorded and associated with the selected utterance name. Before any events are recorded Language Maker checks 254 if Language Maker is recording; if not, Language Maker returns 256. If recording is on 258, then Language Maker checks the current recording mode.

While recording, Language Maker seizes control of the operating system by setting control flags that cause the operating system to call Language Maker every tick of the Macintosh (i.e. every 1/60 second).

If the user has set Language Maker in dialog mode, Language Maker can record dialog events (i.e. events which involve modal dialog, where the user cannot do anything except respond to the actions in modal dialog boxes). To accomplish this, the user must be able to produce actions (i.e. mouse clicks, menu selections) in the current application so that the dialog boxes are prompted to the screen. Then the user can initialize recording and respond to the dialog

T06050" 6402580

boxes. When modal dialog boxes should be produced, events received by Language Maker are also forwarded to the operating system. Otherwise, events are not forwarded to the operating system. Language Maker's modal dialog recording is performed by the Run  
5 Modal module 260.

If modal dialog events are not being recorded, the user records with Language Maker in "action" mode, and Language Maker proceeds to the Run Edit module 262.

10 In the accEvent branch, all events are forwarded to the Event Handler module 264.

In the accMenu branch, the menu indicated by the desk accessory menu event is checked 266. If the event occurred in the Language Maker menu, it is forwarded to the Do My Menu module 268. Other events are ignored 270.

15 Referring to Fig. 5, the Run Edit module 262 performs a loop 272, 274. Each action is recorded by the Record Actions submodule 272. If there are more actions in the event queue then the loop returns to the Record Actions submodule. If a cancel action appears 276 in the event queue then Run Edit returns 277 without  
20 updating the current language in memory. Otherwise, if the events are completed successfully, run edit updates the language in memory and turns off recording 278 and returns to the operating system 280.

25 Referring to Fig. 6, in the Record Actions submodule 272, actions performed by the user in record mode are recorded. When the current application makes a request for the next event on the event queue, the event is checked by record actions. Each non-null event (i.e. each action) is processed by Record Actions. First, the type of action is checked 282. If the action selects a menu  
30 284, then the selected menu is recorded. If the action is a mouse click 286, the In Button? routine (see Fig. 8) checks if the click occurred inside of a button (a button is a menu selection area in the front window) or not. If so, the button is recorded 288. If not, the location of the click is recorded 290.

35 Other actions are recorded by special handlers. These actions

include group actions 292, mouse down actions 294, mouse up actions 296, zoom actions 298, grow actions 300, and next window actions 302.

Some actions in menus can create pop-up menus with subchoices. These actions are handled by popping up the appropriate pop-up menu so that the user may select the desired subchoice. Move actions 304, pause actions 306, scroll actions 308, text actions 310 and voice actions 312 pop up respective menus and Record Actions checks 314 for the menu selection made by the user (with a mouse drag). If no menu selection is made, then no action is recorded 316. Otherwise, the choice is recorded 318.

Other actions may launch applications. In this case 320 the selected application is determined. If no application has been selected then no action is recorded 322, otherwise the selected application is recorded 324.

Referring to Fig. 7, the Run Modal procedure 260 allows recording of the modal dialogs of the Macintosh computer. During modal dialogs, the user cannot do anything except respond to the actions in the modal dialog box. In order to record responses to those actions, Run Modal has several phases, each phase corresponding to a step in the recording process.

In the first phase, when the user selects dialog recording, Run Modal prompts the user with a Language Maker dialog box that gives the user the options "record" and "cancel" (see Fig. 25). The user may then interact with the current application until arriving at the dialog click that is to be recorded. During this phase, all calls to Run Modal are routed through Select Dialog 326, which produces the initial Language Maker dialog box, and then returns 327, ignoring further actions.

To enter the second, recording, phase, the user clicks on the "record" button in the Language Maker dialog box, indicating that the following dialog responses are to be recorded. In this phase, calls to Run Modal are routed to Record 328, which uses the In Button? routine 330 to check if a button in current application's dialog box has been selected. If the click occurred in a button,

then the button is recorded 332, and Run Modal returns 333. Otherwise, the location of the click is recorded 334 and Run Modal returns 335.

5 Finally, when all clicks are recorded, the user clicks on the "cancel" button in the Language Maker dialog box, entering the third phase of the recording session. The click in the "cancel" button causes Run Modal to route to Cancel 336, which updates 338 the current language in memory, then returns 340.

10 Referring to Fig. 8, the In Button? procedure 286 determines whether a mouse click event occurred on a button. In Button? gets the current window control list 342 (a Macintosh global which contains the locations of all of the button rectangles in the current window, refer to Appendix B) from the operating system and parses the list with a loop 344 - 350. Each control is fetched 15 350, and then the rectangle of the control is found 346. Each rectangle is analyzed 348 to determine if the click occurred in the rectangle. If not, the next control is fetched 350, and the loop recurses. If, 344, the list is emptied, then the click did not occur on a button, and no is returned 352. However, if the click 20 did occur in a rectangle, then, if, 351, the rectangle is named, the click occurred on a button, and yes is returned 354; if the rectangle is not named 356, the click did not occur on a button, and no is returned 356.

25 Referring to Fig. 9, the Event Handler module 264 deals with standard Macintosh events in the Language Maker display window. The Language Maker display window lists the utterance names in the current language. As shown in Fig. 9, Event Handler determines 358 whether the event is a mouse or keyboard event and subsequently performs the proper action on the Language Maker window.

30 Mouse events include: dragging the window 360, growing the window 362, scrolling the window 364, clicking on the window 368 (which selects an utterance name), and dragging on the window 370 (which moves an utterance name from one location on the screen to another, potentially changing the utterance's position in the language hierarchy). Double-clicking 366 on an utterance name in 35

the window selects that utterance name for action recording, and therefore starts the Run Edit module.

Keyboard events include the standard cut 372, copy 374, and paste 376 routines, as well as cursor movements down 380, up 382, right 384, and left 386. Pressing return at the keyboard 378, as with a double click at the mouse, selects the current utterance name for action recording by Run Edit. After the appropriate command handler is called, Event Handler returns 388. The modifications to the language hierarchy performed by the Event Handler module are reflected in hierarchical structure of the language file produced by the Write Production module during close and save operations.

Referring to Fig. 10, the Do My Menu module 268 controls all of the menu choices supported by Language Maker. After summoning the appropriate submodule (discussed in detail in Figs. 11A through 11I), Do My Menu returns 408.

Referring to Fig. 11A, the New submodule 390 creates a new language. The New submodule first checks 410 if Language Maker is open. If so, it prompts the user 412 to save the current language as a language file. If the user saves the current language, New calls Write Production module 414 to save the language. New then calls Create Global Words 416 and forms a new language 418. Create Global Words 416 will automatically enter a few global (i.e. resident in all languages) utterance names and command strings into the new language. These utterance names and command strings allow the user to make Voice Control commands, and correspond to utterances such as "show me the active words" and "bring up the voice options" (the utterance macros for the corresponding voice file are trained by the user, or copied from an existing voice file, after the new language is saved).

Referring to Fig. 11B, the Open submodule 392 opens an existing language for modification. The Open submodule 392 checks 420 if Language Maker is open. If so, it prompts the user 422 to save the current language, calling Write Production 424 if yes. Open then prompts the user to open the selected language 426. If

the user cancels, Open returns 428. Otherwise, the language is loaded 430 and Open returns 432.

Referring to Fig. 11C, the Save submodule 394 saves the current language in memory as a language file. Save prompts the user to save the current language 434. If the user cancels, Save returns 436, otherwise, Save calls Write Production 438 to convert the language into a state machine control file suitable for use by VOCAL (Fig. 2). Finally, Save returns 440.

Referring to Fig. 11D, the New Action submodule 396 initializes the event recorders to begin recording a new sequence of actions. New Action initializes the event recorder by displaying an action window to the user 442, setting up a tool palette for the user to use, and initializing recording of actions. Then New Action returns 444. After New Action is started, actions are not delivered to the operating system directly; rather they are filtered through Language Maker.

Referring to Fig. 11E, the Record Dialog submodule 398 records responses to dialog boxes through the use of the Run Modal module. Record Dialog 398 gives the user a way to record actions in modal dialog; otherwise the user would be prevented from performing the actions which bring up the dialog boxes. Record Dialog displays 446 the dialog action window (see Fig. 25) and turns recording on. Then Record Dialog returns 448.

Referring to Fig. 11F, the Create Default Menus submodule 400 extracts default utterance names (and generates associated command strings) from the executable code for an application. Create Default Menus 270 is ordinarily the first choice selected by a user when creating a language for a particular application. This submodule looks at the executable code of an application and creates an utterance name for each menu command in the application, associating the utterance name with a command string that will select that menu command. When called, Create Default Menus gets 450 the menu bar from the executable code of the application, and initializes the current menu to be the first menu (X=1). Next, each menu is processed recursively. When all menus are processed,

Create Default Menus returns 454. A first loop 452, 456, 458, 460 locates the current ( $X^{\text{th}}$ ) menu handle 456, initializes menu parsing, checks if the current menu is fully parsed 458, and reiterates by updating the current menu to the next menu. A second loop 458, 462, 464 finds each menu name 462, and checks 464 if the name is hierarchical (i.e. if the name points to further menus). If the names are not hierarchical, the loop recurses. Otherwise, the hierarchical menu is fetched 466, and a third loop 470, 472 starts. In the third loop, each item name in the hierarchical menu is fetched 472, and the loop checks if all hierarchical item names have been fetched 470.

Referring to Fig. 11G, the Create Default Text submodule 402 allows the user to convert a text file on the clipboard into a list of utterance names. Create default text 402 creates an utterance name for each unique word in the clipboard 474, and then returns 476. The utterance names are associated with the keyboard entries which will type out the name. For example, a business letter can be copied from the clipboard into default text. Utterances would then be associated with each of the common business terms in the letter. After ten or twelve business letters have been converted the majority of the business letter words would be stored as a set of utterances.

Referring to Fig. 11H, the Alphabetize Group submodule 404 allows the user to alphabetize the utterance names in a language. The selected group of names (created by dragging the mouse over utterance names in the Language Maker window) is alphabetized 478, and then Alphabetize Group returns 480.

Referring to Fig. 11I, the Preferences submodule 406 allows the user to select standard graphic user interface preferences such as font style 482 and font size 484. The Preferences submenu 486 allows the user to state the metric by which mouse locations of recorded actions are stored. The coordinates for mouse actions can be relative to the global window coordinates or relative to the application window coordinates. In the case where application menu selections are performed by mouse clicks, the mouse clicks must

T06050" 61025860

5  
10

15

20

25

30

35



current level of the language. Next, the string "TERMINAL =" is written 506, and if, 508, the language level is terminal, the string "(" is written. Next, Write Production checks 512 for sub-levels in the language. If no sub-levels exist, Write Production returns 514. Otherwise, the sub-levels are processed by another call 516 to Write Production on the sub-level of the language. After the sub-level is processed, Write Production writes the string ")" and returns 518.

Referring to Fig. 13, the Write Terminal submodule 496 writes each utterance name and the associated command string to the language file. First, Write Terminal checks 520 if it is at a terminal. If not, it returns 530. Otherwise, Write Terminal writes 522 the string corresponding to the utterance name to the language file. Next, if, 524, there is an associated command string, Write Terminal writes the command string (i.e. "output") to the language file. Finally, Write Terminal writes 528 the string ";" to the language file and returns 530.

#### Voice Control

The Voice Control software serves as a gate between the operating system and the applications running on the operating system. This is accomplished by setting the Macintosh operating system's `get_next_event` procedure equal to a filter procedure created by Voice Control. The `get_next_event` procedure runs when each `next_event` request is generated by the operating system or by applications. Ordinarily the `get_next_event` procedure is null, and `next_event` requests go directly to the operating system. The filter procedure passes control to Voice Control on every request. This allows Voice Control to perform voice actions by intercepting mouse and keyboard events, and create new events corresponding to spoken commands.

The Voice Control filter procedure is shown in Fig. 14.

After installation 538, the `get_next_event` filter procedure 540 is called before an event is generated by the operating system. The event is first checked 542 to see if it is a null event. If so, the Process Input module 544 is called directly. The Process

Input routine 544 checks for new speech input and processes any that has been received. After Process Input, the Voice Control driver proceeds through normal filter processing 546 (i.e., any filter processing caused by other applications) and returns 548.

5 If the next event is not a null event, then displays are hidden 550. This allows Voice Control to hide any Voice Control displays (such as current language lists) which could have been generated by a previous non-null action. Therefore, if any prompt windows have been produced by Voice Control, when a non-null event occurs,

10 the prompt windows are hidden. Next, key down events are checked 552. Because the recognizer is controlled (i.e. turned on and off) by certain special key down events, if the event is a key down event then Voice Control must do further processing. Otherwise, the Voice Control drive procedure moves directly to Process Input

15 544. If a key down event has occurred 554, where appropriate, software latches which control the recognizer are set. This allows activation of the Recognizer Software, the selection of Recognizer options, or the display of languages. Thereafter, the Voice Control driver moves to Process Input 544.

20 Referring to Fig. 15, the Process Input routine is the heart of the Voice Control driver. It manages all voice input for the Voice Navigator. The Process Input module is called each time an event is processed by the operating system. First 546, any latches which need to be set are processed, and the Macintosh waits for a

25 number of delay ticks, if necessary. Delay ticks are included, for example, where a menu drag is being performed by Voice Control, to allow the menu to be drawn on the screen before starting the drag. Also, some applications require delay between mouse or keyboard events. Next, if recognition is activated 548 the process input

30 routine proceeds to do recognition 562. If recognition is deactivated, Process Input returns 560.

The recognition routine 562 prompts the recognition drivers to check for an utterance (i.e., sound that could be speech input). If there is recognized speech input 564, Process Input checks the

35 vertical blanking interrupt VBL handler 566, and deactivates it

T0605049-050901

where appropriate.

The vertical blanking interrupt cycle is a very low level cycle in the operating system. Every time the screen is refreshed, as the raster is moving from the bottom right to the top left of the screen, the vertical blanking interrupt time occurs. During this blanking time, very short and very high priority routines can be executed. The cycle is used by the Process Input routine to move the mouse continuously by very slowly incrementing of the mouse coordinates where appropriate. To accomplish this, mouse move events are installed onto the VBL queue. Therefore, where appropriate, the VBL handler must be deactivated to move the mouse.

Other speech input is placed 568 on a speech queue, which stores speech related events for the processor until they can be handled by the ProcessQ routine. However, regardless of whether speech is recognized, ProcessQ 570 is always called by Process Input. Therefore, the speech events queued to ProcessQ are eventually executed, but not necessarily in the same Process Input cycle. After calling ProcessQ, Process Input returns 571.

Referring to Fig. 16, the Recognize submodule 562 checks for encoded utterances queued by the Voice Navigator box, and then calls the recognition drivers to attempt to recognize any utterances. Recognize returns the number of commands in (i.e. the length of) the command string returned from the recognizer. If, 572, no utterance is returned from the recognizer, then Recognize returns a length of zero (574), indicating no recognition has occurred. If an utterance is available, then Recognize calls sdi\_recognize 576, instructing the Recognizer Software to attempt recognition on the utterance. If, 578, recognition is successful, then the name of the utterance is displayed 582 to the user. At the same time, any close call windows (i.e. windows associated with close call choices, prompted by Voice Control in response to the Recognizer Software) are cleared from the display. If recognition is unsuccessful, the Macintosh beeps 580 and zero length is returned 574.

If recognition is successful, Recognize searches 584 for an

T06050" 64025960

output string associated with the utterance. If there is an output string, recognize checks if it is asleep 586. If it is not asleep 590, the output count is set to the length of the output string and, if the command is a control command 592 (such as "go to sleep" or "wake up"), it is handled by the Process Voice Commands routine 594.

If there is no output string for the recognized utterance, or if the recognizer is asleep, then the output of Recognize is zero (588). After the output count is determined 596, the state of the recognizer is processed 596. At this time, if the Voice Control state flags have been modified by any of the Recognize subroutines, the appropriate actions are initialized. Finally, Recognize returns 598.

Referring to Fig. 17, the Process Voice Commands module deals with commands that control the recognizer. The module may perform actions, or may flag actions to be performed by the Process States block 596 (Fig. 16). If the recognizer is put to sleep 600 or awakened 604, the appropriate flags are set 602, 606, and zero is returned 626, 628 for the length of the command string, indicating to Process States to take no further actions. Otherwise, if the command is scratch\_that 608 (ignore last utterance), first\_level 612 (go to top of language hierarchy, i.e. set the Voice Control state to the root state for the language), word\_list 616 (show the current language), or voice\_options 620, the appropriate flags are set and 610, 614, 618, 622, and a string length of -1 is returned 624, 628, indicating that the recognizer state should be changed by Process States 596 (Fig. 16).

Referring to Fig. 18 the ProcessQ module 570 pulls speech input from the speech queue and processes it. If, 630, the event queue is empty then ProcessQ may proceed, otherwise ProcessQ aborts 632 because the event queue may overflow if speech events are placed on the queue along with other events. If, 634, the speech queue has any events then process queue checks to see if, 636, delay ticks for menu drawing or other related activities have expired. If no events are on the speech queue the ProcessQ aborts

636. If delay ticks have expired, then ProcessQ calls Get Next 642 and returns 644. Otherwise, if delay ticks have not expired, ProcessQ aborts 640.

Referring to Fig. 19, the Get Next submodule 642 gets  
 5 characters from the speech queue and processes them. If, 646, there are no characters in the speech queue then the procedure simply returns 648. If there are characters in the speech queue then Get Next checks 650 to see if the characters are command characters. If they are, then Get Next calls Check Command 660.  
 10 If not, then the characters are text, and Get Next sets the meta bits 652 where appropriate.

When the Macintosh posts an event, the meta bits (see Appendix B) are used as flags for conditioning keystrokes such as the condition key, the option key, or the command key. These keys  
 15 condition the character pressed at the keyboard and create control characters. To create the proper operating system events, therefore, the meta bits must be set where necessary. Once the meta bits are set 652, a key down event is posted 654 to the Macintosh event queue, simulating a keypush at the keyboard.  
 20 Following this, a key up is posted 656 to the event queue, simulating a key up. If, 658, there is still room in the event queue, then further speech characters are obtained and processed 646. If not, then the Get Next procedure returns 676.

If the command string input corresponds to a command rather  
 25 than simple key strokes, the string is handled by the Check Command procedure 660 as illustrated in Figure 19. In the Check Command procedure 660 the next four characters from the speech queue (four characters is the length of all command strings, see Appendix A) are fetched 662 and compared 664 to a command table. If, 666, the  
 30 characters equal a voice command, then a command is recognized, and processing is continued by the Handle Command routine 668. Otherwise, the characters are interpreted as text and processing returns to the meta bits step 652.

In the Handle Command procedure 668 each command is referenced  
 35 into a table of command procedures by first computing 670 the

5

10

35

instead, thereby allowing Menu command to insert the desired menu coordinates in the place of the real coordinates. After posting a mouse down in the appropriate menu bar, Menu Command sets 688 the wait ticks to 30, which gives the operating system time to draw the menu, and returns 690.

In the My Menu Select trap 692 the menuselect global state is reset 694 to clear any previously selected menus, and the desired menu id and the item number are moved to the Macintosh stack 696, thus selecting the desired menu item.

The Find Menu routine 700 collects 702 the command parameters for the desired menu. Next, the menuname is compared 704 to the menu name list. If, 706, there is no menu with the name "menuname", Find Menu exits 708. Otherwise, Find Menu compares 710 the itemname to the names of the items in the menu. If, 712, the located item number is greater than 0, then Find Menu queues 718 the menu id and item number for use by Menu command, and returns 720. Otherwise, if the item number is 0 then Find Menu simply sets 714 the internal Voice Control flags "mousedown" and "global" flags to true. This indicates to Voice Control that the mouse location should be globally referenced, and that the mouse button should be held down. Then Find Menu calls 716 the Post Mouse routine, which references these flags to manipulate the operating system's mouse state accordingly.

Referring to Fig. 21B, the Control command 722 performs a button push within a menu, invoking actions such as the save command in the file menu of an application. To do this, the Control command gets the command parameters 724 from the control string, finds the front window 726, gets the window command list 728, and checks 730 if the control name exists in the control list. If the control name does exist in the control list then the control rectangle coordinates are calculated 732, the Post Mouse routine 734 clicks the mouse in the proper coordinates, and the Control command returns 736. If the control name is not found, the Control command returns directly.

The Keypad command 738 simulates numerical entries at the

T06050" 64025860

Macintosh keypad. Keypad finds the command parameters for the command string 740, gets the keycode value 742 for the desired key, posts a key down event 744 to the Macintosh event queue, and returns 746.

5       The Zoom command 748 zooms the front window. Zoom obtains the front window pointer 750 in order to reference the mouse to the front window, calculates the location of the zoom box 752, uses Post Mouse to click in the zoom box 754, and returns 756.

10       The Local Mouse command 758 clicks the mouse at a locally referenced location. Local Mouse obtains the command parameters for the desired mouse location 760, uses Post Mouse to click at the desired coordinate 762, and returns 764.

15       The Global Mouse command 766 clicks the mouse at a globally referenced location. Global Mouse obtains the command parameters for the desired mouse location 768, sets the global flag to true 770 (to signal to Post Mouse that the coordinates are global), uses Post Mouse to click at the desired coordinate 772, and returns 774.

20       The Double Click command double clicks the mouse at a locally referenced location. Double Click obtains the command parameters for the desired mouse location 778, calls Post Mouse twice 780, 782 (to click twice in the desired location), and returns 784.

25       The Mouse Down command 786 sets the mouse button down. Mouse Down sets the mousedown flag to true 788 (to signal to Post Mouse that mouse button should be held down), uses Post Mouse to set the button down 790, and returns 792.

30       The Mouse Up command 794 sets the mouse button up. Mouse Up sets the mbState global (see Appendix B) to Mouse Button UP 796 (to signal to the operating system that mouse button should be set up), posts a mouse up event to the Macintosh event queue 798 (to signal to applications that the mouse button has gone up), and returns 800.

35       Referring to Fig. 21D, the Screen Down command 802 scrolls the contents of the current window down. Screen Down first looks 804 for the vertical scroll bar in the front window. If, 806, the scroll bar is not found, Screen Down simply returns 814. If the



scroll bar is found, Screen Down calculates the coordinates of the down arrow 808, sets the mousedown flag to true 810 (indicating to Post Mouse that the mouse button should be held down), uses Post Mouse to set the mouse button down 812, and returns 814.

5       The Screen Up command 816 scrolls the contents of the current window up. Screen Up first looks 818 for the vertical scroll bar in the front window. If, 820, the scroll bar is not found, Screen Up simply returns 828. If the scroll bar is found, Screen Up calculates the coordinates of the up arrow 822, sets the mousedown flag to true 824 (indicating to Post Mouse that the mouse button should be held down), uses Post Mouse to set the mouse button down 826, and returns 828.

15       The Screen Left command 830 scrolls the contents of the current window left. Screen Left first looks 832 for the horizontal scroll bar in the front window. If, 834, the scroll bar is not found, Screen Left simply returns 842. If the scroll bar is found, Screen Left calculates the coordinates of the left arrow 836, sets the mousedown flag to true 838 (indicating to Post Mouse that the mouse button should be held down), uses Post Mouse to set the mouse button down 840, and returns 842.

25       The Screen Right command 844 scrolls the contents of the current window right. Screen Right first looks 846 for the horizontal scroll bar in the front window. If, 848, the scroll bar is not found, Screen Right simply returns 856. If the scroll bar is found, Screen Right calculates the coordinates of the right arrow 850, sets the mousedown flag to true 852 (indicating to Post Mouse that the mouse button should be set down), uses Post Mouse to set the mouse button down 854, and returns 856.

30       Referring to Fig. 21E, the Page Down command 858 moves the contents of the current window down a page. Page Down first looks 860 for the vertical scroll bar in the front window. If, 862, the scroll bar is not found, Page Down simply returns 868. If the scroll bar is found, Page Down calculates the page down button coordinates 864, uses Post Mouse to click the mouse button down 866, and returns 868.

09852049-050901  
TOP SECRET

The Page Up command 870 moves the contents of the current window up a page. Page Up first looks 872 for the vertical scroll bar in the front window. If, 874, the scroll bar is not found, Page Up simply returns 880. If the scroll bar is found, Page Up calculates the page up button coordinates 876, uses Post Mouse to click the mouse button down 878, and returns 880.

The Page Left command 882 moves the contents of the current window left a page. Page Left first looks 884 for the horizontal scroll bar in the front window. If, 886, the scroll bar is not found, Page Left simply returns 892. If the scroll bar is found, Page Left calculates the page left button coordinates 888, uses Post Mouse to click the mouse button down 890, and returns 892.

The Page Right command 894 moves the contents of the current window right a page. Page Right first looks 896 for the horizontal scroll bar in the front window. If, 898, the scroll bar is not found, Page Right simply returns 904. If the scroll bar is found, Page Right calculates the page right button coordinates 900, uses Post Mouse to click the mouse button down 902, and returns 904.

Referring to Fig. 21F, the Move command 906 moves the mouse from its current location (y,x), to a new location (y+ $\delta y$ , x+ $\delta x$ ). First, Move gets the command parameters 908, then Move sets the mouse speed to tablet 910 (this cancels the mouse acceleration, which otherwise would make mouse movements uncontrollable), adds the offset parameters to the current mouse location 912, forces a new cursor position and resets the mouse speed 914, and returns 916.

The Move to Global Coordinate command 918 moves the cursor to the global coordinates given by the Voice Control command string. First, Move to Global gets the command parameters 920, then Move to Global checks 922 if there is a position parameter. If there is a position parameter, the screen position coordinates are fetched 924. In either case, the global coordinates are calculated 926, the mouse speed is set to tablet 928, the mouse position is set to the new coordinates 930, the cursor is forced to the new position 932, and Move to Global returns 934.

The Move to Local Coordinate command 936 moves the cursor to the local coordinates given by the Voice Control command string. First, Move to Local gets the command parameters 938, then Move to Local checks 940 if there is a position parameter. If there is a position parameter, the local position coordinates are fetched 942. In either case, the global coordinates are calculated 944, the mouse speed is set to tablet 946, the mouse position is set to the new coordinates 948, the cursor is forced to the new position 950, and Move to Global returns 952.

The Move Continuous command 954 moves the mouse continuously from its present location, moving  $\delta y, \delta x$  every refresh of the screen. This is accomplished by inserting 956 the VBL Move routine 960 in the Vertical Blanking Interrupt queue of the Macintosh and returning 958. Once in the queue, the VBL Move routine 960 will be executed every screen refresh. The VBL Move routine simply adds the  $\delta y$  and  $\delta x$  values to the current cursor position 962, resets the cursor 964, and returns 966.

Referring to Fig. 21G, the Option Key Down command 968 sets the option key down. This is done by setting the option key bit in the keyboard bit map to TRUE 970, and returning 972.

The Option Key Up command 974 sets the option key up. This is done by setting the option key bit in the keyboard bit map to FALSE 976, and returning 978.

The Shift Key Down command 980 sets the shift key down. This is done by setting the shift key bit in the keyboard bit map to TRUE 982, and returning 984.

The Shift Key Up command 986 sets the shift key up. This is done by setting the shift key bit in the keyboard bit map to FALSE 988, and returning 990.

The Command Key Down command 992 sets the command key down. This is done by setting the command key bit in the keyboard bit map to TRUE 994, and returning 996.

The Command Key Up command 998 sets the command key up. This is done by setting the command key bit in the keyboard bit map to FALSE 1000, and returning 1002.

09852049-050901  
T06050-64025960

The Control Key Down command 1004 sets the control key down. This is done by setting the control key bit in the keyboard bit map to TRUE 1006, and returning 1008.

5 The Control Key Up command 1010 sets the control key up. This is done by setting the control key bit in the keyboard bit map to FALSE 1012, and returning 1014.

The Next Window command 1016 moves the front window to the back. This is done by getting the front window 1018 and sending it to the back 1020, and returning 1022.

10 The Erase command 1024 erases numchars characters from the screen. The number of characters typed by the most recent voice command is stored by Voice Control. Therefore, Erase will erase the characters from the most recent voice command. This is done by a loop which posts delete key keydown events 1026 and checks  
15 1028 if the number posted equals numchars. When numchars deletes have been posted, Erase returns 1030.

The Capitalize command 1032 capitalizes the next keystroke. This is done by setting the caps flag to TRUE 1034, and returning 1036.

20 The Launch command 1038 launches an application. The application must be on the boot drive no more than one level deep. This is done by getting the name of the application 1040 ("appl\_name"), searching for appl\_name on the boot volume 1042, and, if, 1044, the application is found, setting the volume to the  
25 application folder 1048, launching the application 1050 (no return is necessary because the new application will clear the Macintosh queue). If the application is not found, Launch simply returns 1046.

Referring to Fig. 22, the Post Mouse routine 1052 posts mouse  
30 down events to the Macintosh event queue and can set traps to monitor mouse activity and to keep the mouse down. The actions of Post Mouse are determined by the Voice Control flags global and mousedown, which are set by command handlers before calling Post Mouse. After a Post Mouse, when an application does a  
35 get\_next\_event it will see a mouse down event in the event queue,

"050901" 050901

First, Post Mouse saves the current mouse location 1054 so that the mouse may be returned to its initial location after the mouse events are produced. Next the cursor is hidden 1056 to shield the user from seeing the mouse moving around the screen. Next the global flag is checked. If, 1058, the coordinates are local (i.e. global = FALSE) then they are converted 1060 to global coordinates. Next, the mouse speed is set to tablet 1062 (to avoid acceleration problems), and the mouse down is posted to the Macintosh event queue 1064. If, 1066, the mousedown flag is TRUE (i.e. if the mouse button should be held down) then the Set Mouse Down routine is called 1072 and Post Mouse returns 1070. Otherwise, if the mouse down flag is FALSE, then a click is created by posting a mouse up event to the Macintosh event queue 1068 and returning 1070.

The My Button trap 1084 replaces the Macintosh button trap, thereby seizing control of the button state from the operating system. Each time My Button is called, it checks 1086 the

Macintosh mouse button state bit mbState. If mbState has been set to UP, My Button moves to the End Button routine 1106 which sets mbState to UP 1108, removes any VBL routine which has been installed 1110, resets the Button and Post Event traps to the original Macintosh traps 1112, resets the mouse speed and couples the cursor to the mouse 1114, shows the cursor 1102, and returns 1104.

However, if the mouse button is to remain down, My Button checks for the expiration of wait ticks (which allow the Macintosh time to draw menus on the screen) 1088, and calls the recognize routine 1090 to recognize further speech commands. After further speech commands are recognized, My Button determines 1092 its next action based on the length of the command string. If the command string length is less than zero, then the next voice command was a Voice Control internal command, and the mouse button is released by calling End Button 1106. If the command string length is greater than zero, then a command was recognized, and the command is queued onto the voice que 1094, and the voice queue is checked for further commands 1096. If nothing was recognized (command string length of zero), then My Button skips directly to checking the voice queue 1096. If there is nothing in the voice queue, then My Button returns 1104. However, if there is a command in the voice queue, then My Button checks 1098 if the command is a mouse movement command (which would cause a mouse drag). If it is not a mouse movement, then the mouse button is released by calling End Button 1106. If the command is a mouse movement, then the command is executed 1100 (which drags the mouse), the cursor is displayed 1102, and My Button returns.

#### Screen Displays

Referring to Fig. 24, a screen display of a record actions session is shown. The user is recording a local mouse click 1106, and the click is being acknowledged in the action list 1108 and in the action window 1110.

Referring to Fig. 25, a record actions session using dialog boxes is shown. The dialog boxes 1112 for recording a manual

T06050" 54025860

printer feed are displayed to the user, as well as the Voice Control Run Modal dialog box 1114 prompting the user to record the dialogs. The user is preparing to record a click on the Manual Feed button 1116.

5 Referring to Fig. 26, the Language Maker menu 1118 is shown.

Referring to Fig. 27, the user has requested the current language, which is displayed by Voice Control in a pop-up display 1120.

10 Referring to Fig. 28, the user has clicked on the utterance name "apple" 1122, requesting a retraining of the utterance for "apple". Voice Control has responded with a dialog box 1124 asking the user to say "apple" twice into the microphone.

15 Referring to Fig. 29, the text format of a Write Production output file 1126 (to be compiled by VOCAL) and the corresponding Language Maker display for the file 1128 are shown. It is clear from Fig. 29 that the Language Maker display is far more intuitive.

Referring to Fig. 30, a listing of the Write Production output file as displayed in Fig. 29 is provided.

#### Other Embodiments

20 Other embodiments of the invention are within the scope of the claims which follow the appendices. For example, the graphic user interface controlled by a voice recognition system could be other than that of the Apple Macintosh computer. The recognizer could be other than that marketed by Dragon Systems.

25 Included in the Appendices are Appendix A, which sets forth the Voice Control command language syntax, Appendix B, which lists some of the Macintosh OS globals used by the Voice Navigator system, Appendix C, which is a fiche of the Voice Navigator executable code, Appendix D, which is the Developer's Reference  
30 Manual for the Voice Navigator system, and Appendix E, which is the Voice Navigator User's Manual, all incorporated by reference herein.

A portion of the disclosure of this patent document contains material which is subject to copyright protection (for example,  
35 the microfiche Appendix, the User's Manual, and the Reference

T06050" 6402580

[illegible]



## Appendix A: Voice Control Command Language Syntax

Menu Command - @MENU(menuname,itemnum).

Finds item named itemnum in the menu named menuname and selects it. If itemnum is 0, hold the menu down.

5        Control Command - @CTRL(ctlname)

Finds the control named ctlname and clicks in its rectangle.

Key Pad Command - @KYPD(n), where n = 0-9, -, +, \*, /, =, and c for clear

Posts a Keydown for keys on the numeric keypad.

10       Zoom Command - @ZOOM

Clicks in the zoom box of the front window.

Local Mouse Click Command - @LMSE(y,x)

Clicks at local coordinates (y,x) of the front window.

Global Mouse Click Command - @GMSE(y,x)

15       Clicks at the global coordinates (y,x) of the current screen.

Double Click Command - @DCLK(y,x)

Double clicks at the global coordinates (y,x) of the current screen. If y=x=0, double click at the current Mouse location.

Mouse Down Command - @MSDN

20       Set the mouse button state to down and set up traps to keep it down.

Mouse Up Command - @MSUP

Set the mouse button state to up.

Scroll Down Command - @SCDN

25       Post a mouse down in the down arrow portion of the front

T06050" 64025360

window's scroll bar.

Scroll Up Command - @SCUP

Post a mouse down in the up arrow portion of the front window's scroll bar.

5      Scroll Left Command - @SCUP

Post a mouse down in the left arrow portion of the front window's scroll bar.

Scroll Right Command - @SCRT

10      Post a mouse down in the right arrow portion of the front window's scroll bar.

Page Down Command - @PGDN

Click in the page down portion of the front window's scroll bar.

Page Up Command - @PGUP

15      Click in the page up portion of the front window's scroll bar.

Page Left Command - @PGLF

Click in the page left portion of the front window's scroll bar.

Page Right Command - @PGRT

20      Click in the page right portion of the front window's scroll bar.

Move Command - @MOVE( $\delta y, \delta x$ )

25      Move the Mouse from its current location ( $y, x$ ), to a new location ( $y+\delta y, x+\delta x$ ) where  $\delta y$  and  $\delta x$  are pixels and can be either positive or negative values.

Move Continuous Command - MOVI( $\delta y, \delta x$ )

09852049-0509001

Move the mouse continuously from its present location, moving  $\delta y, \delta x$  every refresh of the screen.

Move to Local Coordinate Command - `MOVL(y,x,<windowname>)` or `MOVL(n,<y,x,<windowname>>)` where  $n=N,S,E,W,NE,SE,SW,NW,C,G$

5      Move the cursor to the local coordinates given by  $(y,x)$  or by  $(n.v+y,n.h+x)$ . Use the `grafPort` of the window named "windowname".  
If there is no "windowname" use the `grafPort` of the front window.

Move to Global Coordinate Command - `@MOVG(n,<y,x>)`  
where  $n=N,S,E,W,NE,SE,SW,NW,C,G$

10      move the cursor to the global coordinates given by  $(y,x)$  or by  $(n.v+y,n.h+x)$ . Use the `grafPort` of the screen.

Option Key Down Command - `@OPTD`  
Press (and hold) the option key.

Option Key Up Command - `@OPTU`  
15      Release the option key.

Shift Key Down Command - `@SHFD`  
Press (and hold) the shift key.

Shift Key Up Command - `@SHFU`  
Release the shift key.

20      Command Key Down Command - `@CMDD`  
Press (and hold) the command key.

Command Key Up Command - `@CMDU`  
Release the command key.

25      Control Key Down Command - `@CTLD`  
Press (and hold) the control key.

T06050" 64025860

Control Key Up Command - @CTLU

Release the control key.

Next Window Command - @NEXT

Sends the front window to the back.

5

Erase Command - @ERAS

Erase the last numChars typed.

Capitalize Command - @CAPS

Capitalize the next letter typed.

Launch Command - @LAUN(application\_name)

10

Launch the application named application\_name. The application must be on the boot drive no more than one level deep.

Wait Command - @WAIT(nnn)

Wait for nnn ticks to elapse before doing anything else in recognition.

T06050"64025850

Appendix B: Macintosh OS Globals

Interfacing to the Macintosh Operating System requires that certain low memory globals be managed by Voice Control. The following describes the most important globals. Further  
 5 information is available in "Inside Macintosh", Vols. I-V.

Mouse Globals

MickeyBytes EQU \$D6A - a pointer to the cursor value; used to control the acceleration of the mouse. Set to point to tablet whenever the mouse is moved more than 10 pixels. [pointer]

10 MTemp EQU \$828 - a low-level interrupt mouse location; used to move the mouse during VBL handling while executing a @MOVI command. [long]

Mouse EQU \$830 - the processed mouse coordinate; used to move the mouse for all other @MOVx commands. [long]

15 MBState EQU \$172 - current mouse button state; used to set the MouseDown for @MSDN and for @MENU when itemname = 0. [byte]

Keyboard Globals

20 KeyMap EQU \$174 - keyboard bit map, with one bit mapped to each key on the keyboard. Set the bit to TRUE to set the Meta keys (option, command, shift, control) down. [2 longs]

Filter Globals

JGNEFilter EQU \$29A - Get Next Event filter proc; set to Voice Control's main loop to intercept calls to Get Next Event. [pointer]

Event Queue Globals

25 evtMax EQU \$1E - maximum number of events in the event queue. When this number is reached, stop Posting events.

09852049 050901

EventQueue EQU \$14A - event queue header, the location of the Macintosh event queue. [10 bytes]

#### Time Globals

5 Ticks EQU \$16A - Tick count, time since boot. Used to measure elapsed time between Voice Control actions. [long]

#### Cursor Globals

CrsrCouple EQU \$8CF - cursor coupled to mouse? Used to disconnect cursor when doing remote clicks with @LMSE and @GMSE. [byte]

10 CrsrNew EQU \$8CE - Cursor changed? Force a new cursor after moving the cursor. [byte]

#### Menu Globals

15 MenuList EQU \$A1 Current menuBar list structure. This handle can be de-referenced to find all the menus associated with an application. Use for @MENU commands [handle]

#### Window Globals

20 WindowList EQU \$9D6 - Z-ordered linked list of windows. This pointer will lead to a chain of all existing windows for an application. Use to find a window queue for all local commands. [pointer]

#### Window Offsets

These values are offsets within the window records that describe characteristics of the window. Once a window is located, these offsets are used to calculate:

25 thePort EQU 0 - GrafPtr; local coordinates for @LMSE and @MOVL commands.

portRect EQU \$10 - port's rectangle [rect]; window relative forms of the @MOVL command.

controlList EQU 140 - used to find the controls associated

09852049-050901  
T06050-64025860

with a window.

controlTitle EQU 40 - used to compare control Titles for @CTRL commands.

5 controlRect EQU 8 - used to calculate the click locations in a control.

nextWindow EQU 144 - used to locate the next window for the @NEXT command.

T06050" 64025350

Claims

1           1. A system for enabling voiced utterances to be substituted  
2 for manipulation of a pointing device, the pointing device being  
3 of the kind which is manipulated to control motion of a cursor on  
4 a computer display and to indicate desired actions associated with  
5 the position of the cursor on the display, the cursor being moved  
6 and the desired actions being aided by an operating system in the  
7 computer in response to control signals received from the pointing  
8 device, the computer also having an alphanumeric keyboard, the  
9 operating system being separately responsive to control signals  
10 received from the keyboard in accordance with a predetermined  
11 format specific to the keyboard, the system comprising

12           a voice recognizer for recognizing a voiced utterance, and  
13           an interpreter for converting the voiced utterance into  
14 control signals which will directly create a desired action aided  
15 by the operating system in the computer without first being  
16 converted into control signals expressed in the predetermined  
17 format specific to the keyboard.

1           2. A method for converting voiced utterances to commands,  
2 expressed in a predefined command language, to be used by an  
3 operating system of a computer, comprising

4           converting some voiced utterances into commands corresponding  
5 to actions to be taken by said operating system, and

6           converting other voiced utterances into commands which carry  
7 associated text strings to be used as part of text being processed  
8 in an application program running under said operating system.

TOP SECRET



1           3. A method of generating a table for aiding the conversion  
2 of voiced utterances to commands for use in controlling an  
3 operating system of a computer to achieve desired actions in an  
4 application program running under the operating system, said  
5 application program including menus and control buttons, said  
6 method comprising

7           parsing the instruction sequence of the application program  
8 to identify menu entries and control buttons, and

9           including in said table an entry for each menu entry and  
10 control button found in said application program, each said entry  
11 containing a control command corresponding to said menu entry or  
12 control button.

1           4. A method of enabling a user to create an instance in a  
2 formal language of the kind which has a strictly defined syntax,  
3 comprising

4           providing a graphically displayed list of entries which are  
5 expressed in a natural language and which do not comply with said  
6 syntax,

7           permitting the user to point to an entry on said list, and  
8           automatically generating said instance corresponding to the  
9 identified entry in the list in response to said pointing.

106050" 64025860

### Abstract of the Disclosure

Voice utterances are substituted for manipulation of a pointing device, the pointing device being of the kind which is manipulated to control motion of a cursor on a computer display and to indicate desired actions associated with the position of the cursor on the display, the cursor being moved and the desired actions being aided by an operating system in the computer in response to control signals received from the pointing device, the computer also having an alphanumeric keyboard, the operating system being separately responsive to control signals received from the keyboard in accordance with a predetermined format specific to the keyboard; in the system, a voice recognizer recognizes the voiced utterance, and an interpreter converts the voiced utterance into control signals which will directly create a desired action aided by the operating system without first being converted into control signals expressed in the predetermined format specific to the keyboard. In another aspect, voiced utterances are converted to commands, expressed in a predefined command language, to be used by an operating system of a computer, by converting some voiced utterances into commands corresponding to actions to be taken by the operating system, and converting other voiced utterances into commands which carry associated text strings to be used as part of text being processed in an application program running under the operating system. In another aspect, a table is generated for aiding the conversion of voiced utterances to commands for use in controlling an operating system of a computer to achieve desired actions in an application program running under the operating system, the application program including menus and control buttons; the instruction sequence of the application program is parsed to identify menu entries and control buttons, and an entry is included in the table for each menu entry and control button found in the application program, each entry in the table containing a command corresponding to the menu entry or control button. In another aspect, a user is enabled to create an instance in a formal language of the kind which has a strictly defined syntax; a graphically displayed list of entries are expressed in

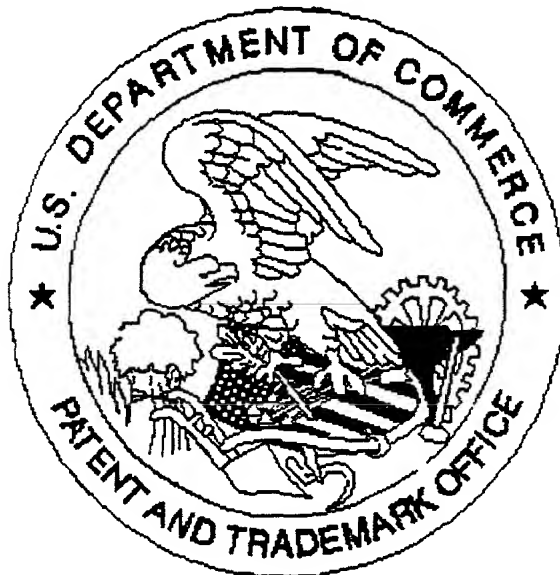
09852049 "050901

a natural language which does not comply with the syntax, the user is permitted to point to an entry on the list, and the instance corresponding to the identified entry in the list is automatically generated in response to the pointing.

09852049-03090  
T06050" 64025860

# United States Patent & Trademark Office

Office of Initial Patent Examination -- Scanning Division



Application deficiencies found during scanning:

☐ Page(s) \_\_\_\_\_ of \_\_\_\_\_ were not present  
for scanning. (Document title)

☐ Page(s) \_\_\_\_\_ of \_\_\_\_\_ were not present  
for scanning. (Document title)

\* Scanned copy is best available. *PAGE 705 IS TORN.*